Variables d'induction généralisées pour l'analyse par instances de programmes récursifs

Pierre Amiranoff, Albert Cohen, Paul Feautrier

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Variables d'induction généralisées pour l'analyse par instances de programmes récursifs

Pierre Amiranoff — Albert Cohen — Paul Feautrier

N° 4252

13 septembre 2001

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4252--FR

Variables d'induction généralisées pour l'analyse par instances de programmes récursifs

Pierre Amiranoff , Albert Cohen , Paul Feautrier

Thème 1 — Réseaux et systèmes
Projet A3

Rapport de recherche n° 4252 — 13 septembre 2001 — 53 pages

Résumé : La sophistication croissante des microprocesseurs et des architectures ouvre la voie de nouvelles techniques d'optimisation dont il est souhaitable de décharger le programmeur. La parallélisation automatique consiste en une phase préalable d'analyse des dépendances, suivie de l'extraction du parallélisme puis de la génération du code parallèle. Nous nous attachons à des programmes, éventuellement récursifs, qui manipulent des structures de données combinant arbres et tableaux.

Ce rapport a pour objet le calcul statique, c.-à-d. lors de la compilation, des adresses dans les structures de données. Ces adresses sont les informations de base pour l'analyse de dépendances. Nous introduisons le concept de *variable d'induction généralisée*, qui étend la notion classique attachée aux nids de boucles. Ce concept formalise le suivi des adresses au travers d'un balisage original de l'exécution.

Ainsi, nous dégageons la notion d'*instance* pour nommer avec exactitude une instruction au cours de l'exécution. L'*analyse par instances* conjugue la finesse des *traces d'exécution* et la richesse du formalisme des langages réguliers.

Nous présentons ici un modèle de programme qui permet le calcul exact de l'adressage pour chaque *instance*. Deux techniques sont exposées, en étroite référence à la théorie des Automates Finis. La première est développée sous forme matricielle, tandis que la seconde fait appel à la théorie des Transducteurs.

Mots-clés : parallélisation automatique, programmes récursifs, analyse de dépendances, analyse par instances, variables d'induction

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

Generalized Induction Variables for Instancewise Analysis of Recursive Programs

Abstract: The increasing sophistication of microprocessors and architectures paves the way for new optimization techniques of which it is desirable to discharge the programmer. Automatic parallelization requires a dependence analysis phase in order to extract parallelism and to generate the derived code. We stick to possibly recursive programs which handle data structures combining trees and arrays.

This report aims at static computation, i.e. during compilation, of addresses in data structures. These addresses are the key information for dependence analysis. We introduce the concept of *generalized induction variable*, which extends the traditional concept relative to nested loops. This concept formalizes the follow-up of addresses through an original beaconing of the execution.

Thus, we introduce the concept of *instance* in order to name every instruction during execution. *Instancewise analysis* builds on the acuracy of *execution traces* and on the richness of the regular languages formalism.

We study a program model which allows the exact computation of addressing for every *instance*. Two techniques are exposed, in narrow reference to the theory of Finite-State Automata. The first one is presented in a matrix form, while the second calls upon the theory of Transducers.

Key-words: automatic parallelization, recursive programs, dependance analysis, instancewise analysis, induction variables

Table des matières

Remerciements, ...

... tout particulièrement à *Véronique Donzeau-Gouge* et *Catherine Dubois* pour le suivi régulier de ce travail, accompagné de judicieux conseils!

Première partie

L'Analyse de programmes par instances

Chapitre 1

Présentation générale

1.1 L'Analyse de programmes

1.1.1 L'Analyse statique

A l'aide des microprocesseurs et des ordinateurs modernes, il est possible d'obtenir des performances de calcul élevées en proportion du degré de parallélisme du programme. L'analyse d'un programme séquentiel permet de mettre en évidence le parallélisme effectif qu'il contient, mais aussi le parallélisme potentiel que l'on est susceptible de révéler par transformation de programme. Selon l'une ou l'autre de ces méthodes, les techniques de parallélisation automatique associées au compilateur vont générer un code parallèle qui prend en compte l'architecture de la machine. *L'Analyse Statique de programmes* a pour but de découvrir des propriétés des programmes à la compilation, c.-à-d. en analysant le texte source. Ces résultats sont en général approchés vu que le comportement exact des programmes est indécidable.

1.1.2 Instances

Une *instruction* du texte source a pour fonction de prescrire une modification de l'état-mémoire de la machine lors d'une exécution du programme. On supposera dans la suite que chaque instruction est étiquetée par un **label** distinctif tiré d'un alphabet noté Σ_{ctrl}.

Au cours d'une exécution particulière e du programme, une même instruction peut être exécutée un grand nombre de fois. Supposons donné un état-mémoire préexistant à cette exécution, finie ou infinie, entière ou partielle, du programme. Cette *exécution* e peut être définie soit comme une suite de configurations de l'état-mémoire, soit comme une liste d'exécutions d'*instruction* (la *trace d'exécution*), chacune assurant la transition entre 2 configurations successives. La liste des **labels** correspondants est appelée *mot de trace*. On peut dénoter de manière non ambiguë l'exécution particulière d'une *instruction* s au sein de e par le *mot de trace* dont le **label** de s est le dernier élément. Cette exécution individuelle sera appelée une *occurrence*.

La notion d'*instance* est plus abstraite que celle *d'occurrence*, comme nous le verrons dans ce rapport. Cependant, les 2 notions peuvent être confondues si l'on s'attache à ne considérer que les *occurrences* d'une exécution particulière du programme.

1.1.3 Analyse de dépendances

L'Analyse de dépendances rassemble des informations sur les *occurrences* d'une exécution de programme qui accèdent à la même cellule-mémoire. La comparaison de 2 *occurrences* appartenant à 2 exécutions différentes n'a pas vraiment de sens; c'est au sein d'une même exécution que la notion d'ordre entre les *occurrences* présente un intérêt. L'ordonnancement de ces accès doit respecter la sémantique spécifiée par le programme-source. La parallélisation d'un programme est donc intéressée au premier chef par *l'Analyse de dépendances*. Il importe de garder à l'esprit que les valeurs contenues dans les cellules-mémoire sont hors du domaine de cette analyse et donc de ce rapport, qui ne s'intéressera qu'aux adresses.

C'est au niveau de finesse de l'*occurrence* que cherche à travailler *l'Analyse de Programmes par Instances*, contrairement aux analyses classiques qui se contentent du degré de finesse de l'instruction-source c.-à-d. du *point de programme*.

Ce rapport se place dans le cadre du modèle formel développé dans la thèse d'Albert Cohen [Coh99], apparenté aux *analyses de programme par instances*. Cette thèse ouvre la voie des analyses par instances pour des programmes comportant des appels de procédures, y compris récursifs.

1.2 Simplifier les mots de trace

1.2.1 Motivation

A priori, la finesse des *occurrences* impose de travailler sur les *traces d'exécution*. En pratique, cela soulève des difficultés rédhibitoires.
Pour un programme donné, l'ensemble des exécutions possibles est en général inconnu à la compilation. En effet, le résultat d'un test conditionnel n'apparaît qu'à l'exécution. Par ailleurs, même si l'on se limite aux boucles *for*, le résultat d'un test d'entrée dans une itération est également inconnu s'il dépend de valeurs calculées à l'exécution, et le nombre d'itérations exécutées reste alors indéterminé à la compilation.

L'ensemble des *traces* est donc inconnu et nous faisons le choix conservatif de travailler sur l'ensemble de toutes les exécutions possibles quel que soit le résultat des tests: lors de tout test conditionnel, les 2 branches sont supposées accessibles, et l'exécution d'une boucle peut conduire à un nombre non borné d'itérations.

Les *mots de trace* peuvent être très longs, voire infinis pour un programme qui ne termine pas. Il serait tentant de les décrire *en intension* plutôt qu'*en extension*, vu la redondance d'informations inhérente à ce modèle: par exemple, si une *occurrence* o_2 suit une *occurrence* o_1 au cours d'une exécution, le *mot de trace* de o_1 préfixe celui de o_2.

1.2.2 Vers un langage rationnel

L'utilisation d'un langage rationnel paraît répondre à nos attentes. Cependant, la dénotation des *mots de trace* dans un tel langage est illusoire: un appel récursif de procédure doit mémoriser les appels antérieurs pour effectuer correctement les retours d'appels. Or un automate à états fini n'a pas cette capacité de mémorisation. Non reconnaissable par un tel automate, le langage des *mots de trace* n'est pas rationnel.

En dépit de ces difficultés, la démarche suivie par [Coh99] maintient l'exigence du calcul des adresses-mémoire en toute *occurrence* d'une exécution. Son originalité est d'attacher certaines contraintes au modèle de programme afin de compresser la dénotation d'une *occurrence*. Le *mot de contrôle*, simplification correspondante du *mot de trace*, s'exprime alors en un *langage rationnel*. Nous montrerons comment construire *l'automate de contrôle*, automate à états fini qui reconnaît ce langage.

Nous supposons connues les bases de la théorie des langages et des automates Finis [HU79, RS97, Aut94].

1.2.3 Structure du programme et adressage-mémoire

Calcul de l'adressage

Le prélude à *l'Analyse de dépendances par instances* est de déterminer les valeurs d'adresse à l'issue de chaque *occurrence*. Dans le cas général, ce calcul est approché, il regroupe l'ensemble des valeurs possibles compte-tenu de la puissance de *l'Analyse Statique*. Cela se traduit par une perte d'informations sur l'historique de l'*occurrence* et donc sur l'évolution de l'adressage-mémoire, lors de la réduction du *mot de trace* en un *mot de contrôle*.

Calcul exact

Dans les limites du présent rapport, nous faisons le choix de présenter des méthodes de calcul exact de l'adressage-mémoire. C'est pourquoi nous avons pris le parti d'imposer au sein même de la syntaxe abstraite les contraintes sur le modèle de programme destinées à garantir cette exactitude. La finalité est de faire coïncider la simplification des *mots de trace* en *mots de contrôle* avec le suivi des modifications de l'adressage-mémoire.

La modélisation

- **bloc:** Le texte source est structuré sous forme de *blocs*. Les *blocs* sont organisés selon deux dimensions orthogonales: la relation d'emboîtement et la relation de séquentialité.
- **Visibilité:** Une variable est représentée par un *identificateur* `v` auquel est liée sa valeur dans une région déterminée du texte source R. La région R est appelée la *visibilité* de la variable `v`.

- **Variables d'induction:** Ce sont des variables distinguées. Elles ont pour fonction de porter en exclusivité l'adressage-mémoire au sein des structures de données: connaître les *variables d'induction* en un point de l'exécution, c'est connaître l'adressage à cet instant. Comme toute autre variable, une *variable d'induction* a une *visibilité* déterminée.
- **Règle des blocs:** Cette règle se décompose en plusieurs critères:
  - 1. Deux *blocs* ne se chevauchent pas; soit ils sont séparés, soit l'un englobe l'autre.
  - 2. la *visibilité* d'une *variable d'induction* i est circonscrite à un *bloc* B qui l'initialise en la déclarant; les *blocs* qui initialisent une *variable d'induction* sont:
    - un *bloc* qui délimite une boucle
    - un *bloc* qui délimite une itération de boucle;
    - un appel de procédure, récursif ou non, est un *bloc* constitué d'une seule instruction;
    - un *bloc* noté *Bloc* dans notre syntaxe, dont la première instruction est la déclaration d'une *variable d'induction*, initialisée selon les contraintes auxquelles il a été fait allusion en Section 1.2.2 et rappelées ci-desssous.
  - 3. l'initialisation de la *variable d'induction* `v` est soit l'affectation à une constante, soit déterminée par une opération dont les arguments sont pris parmi les valeurs des *variables d'induction* vivantes à cet instant; parmi ces variables, il peut en exister une homonyme de `v`. Le nom de variable est alors surchargé (*surcharge dynamique*).
  - 4. `v` est une **constante** quand l'exécution parcourt B.

Intuition de la modélisation

En une *occurrence* donnée o pour laquelle `v` est vivante, sa valeur v est celle qui lui a été donnée à l'ouverture de B. Par conséquent, l'adressage en o est déterminée par l'ensemble des ouvertures de *bloc* qui ont initialisé des *variables d'induction* toujours vivantes en o.

En vertu du critère 3 de la **Règle des blocs**, les valeurs des *variables d'induction* se calculent par héritage: le calcul de v hérite ses arguments des valeurs des *variables d'induction* encore vivantes déclarées plus tôt.

La boucle, que notre syntaxe modélise comme un emboîtement des *blocs* que sont les itérations, est un exemple de *surcharge dynamique*, de même qu'un appel récursif de procédure.

Le calcul de l'adressage, sous la forme d'une fonction dite *de liaison*, est à rapprocher de la notion de variable manipulée par les langages fonctionnels. Le nom `v` de la *variable d'induction* est *liée* à une valeur v qui lui est imposée par *l'environnement*. L'évolution de ce dernier dans les limites de la région R est sans influence sur la *liaison* de v à `v`.

Le qualificatif *variable* dans le terme *variable d'induction* se réfère finalement au passage des **valeurs** à un *nom*, au cours de l'exécution du programme, ce nom étant susceptible de subir une *surcharge dynamique*. C'est pourquoi dans la suite, nous parlerons de *variable d'induction* pour dénoter un *nom*, c'est-à dire un identificateur du texte source.

Domaine de validité du modèle

Tous langages confondus, de nombreux programmes s'adapteront sans difficulté, après une éventuelle transformation, à cette contrainte de la syntaxe abstraite. A titre d'exemple, les transformations du type *code motion* [KRS94, Gup98] ou de *l'exécution symbolique* [Muc97] ont le pouvoir de déplacer une affectation dans la boucle ou la procédure englobante, afin de la transformer en l'initialisation d'une *variable d'induction*. L'extension du modèle à d'autres programmes est laissée pour de futurs travaux. Pour l'heure, ces programmes peuvent être traités en affectant, pour les *occurrences* où une *variable d'induction* modifiée est vivante, la *valeur_indéterminée* à cette variable, ce qui ce répercute aussi sur les *variables d'induction* qui dépendent d'elle.

Un langage rationnel

La finalité de cette modélisation est de structurer l'historique de l'adressage à l'image d'une *pile d'exécution* [ASU86], où l'ouverture de *bloc* jouerait le rôle d'un appel de procédure. De la même manière que, lors d'un retour d'appel, le compilateur "oublie" la liaison temporaire entre le nom d'une variable locale et la valeur qui lui a été associée, le *mot de contrôle*, simplification du *mot de trace*, va "oublier" la *trace* des instructions exécutées lors de l'activation d'un *bloc* qui a été refermé.

La nouvelle dénotation d'une *occurrence oublie* ainsi l'historique des *variables d'induction* qui ne sont plus vivantes à l'issue de l'*occurrence*. Cela permet de l'affilier à un **langage rationnel** tout en gardant toute l'information sur les *variables d'induction* **vivantes** de l'*occurrence*.

L'asservissement de la valeur (des valeurs dans le cas général) d'une *variable d'induction* en une *occurrence* au *mot de contrôle* associé impose des **restrictions sur les opérations** de modification de ces variables. L'opération de base sur les mots d'un langage rationnel est la concaténation. Les opérations que notre modèle offre aux *variables d'induction* devront faire appel à ce seul mécanisme. La fonction dite *de liaison* sera alors calculable dans le cadre de la théorie des *automates Finis* et des *Transducteurs*.

Chapitre 2

Le modèle de programme

2.1 Les structures de données

2.1.1 Les données scalaires et les enregistrements

Le contenu des cellules-mémoire nous étant indifférent, notre analyse s'applique quel que soit le type de base des données mémorisées. A titre d'illustration, notre modèle s'appuie sur les ensembles prédéfinis suivants: les entiers naturels ou relatifs, les caractères d'un alphabet fini, les chaînes de caractères et les booléens, selon les besoins. Il inclut tout opérateur prédéfini sur ces ensembles, ainsi que les types des constantes, variables et expressions.

Nous supposons sans perte de généralité que les structures de données constantes et variables sont déclarées globalement au début du programme.

Une *référence* ou *référence-mémoire* est une expression qui dénote une adresse en mémoire.

L'adresse d'une variable scalaire, qui est constante, ne concerne pas *l'Analyse de dépendances*. Le cas d'un enregistrement de champs de scalaires ou d'enregistrements s'y ramène à condition de "l'aplatir" et de le considérer comme un tout indivisible. Pour la suite, nous assimilons un enregistrement à un scalaire, référencé par une unique adresse.

2.1.2 les données structurées

Ce sont des données dont les composants de base sont accessibles par intermédiaire d'une référence représentative d'une adresse vers la mémoire, référence portée par une variable ou une constante. Nous autorisons les structures suivantes:

- les tableaux de scalaires
- les arbres de scalaires
- les tableaux d'arbres
- les arbres de tableaux

Les tableaux de tableaux sont représentés par des tableaux, et les arbres d'arbres par des arbres.

Nous appellerons *étage* chacune des structures, tableau ou arbre, emboîtées entre elles.

Une référence de tableau est donnée par le vecteur des indices d'un élément de tableau, une référence d'arbre par la chaîne de caractères qui étiquète un chemin depuis la racine.

Une cellule de tableau d'arbres est référencée par la concaténation des adresses de chaque *étage*: l'adresse dans le tableau relativement à la case de tête, exprimée par un vecteur d'entiers, suivie de l'adresse dans l'arbre relativement à la racine, exprimée par une chaîne de caractères. Ce procédé de composition se poursuit naturellement par induction structurelle. Le même principe s'applique pour pour toute qutre structure de données mixte.

A titre d'exemple, considérons le cas d'un tableau nommé TAT (T pour tableau, A pour arbre), de 4×10 arbres binaires de tableaux de 7 scalaires. La référence R_t d'un des scalaires est $\binom{2}{9}rl3$.

Ce scalaire est l'élément n^o3 du tableau situé dans le nœud-fils gauche du nœud-fils droit de la racine de l'arbre placé dans l'élément de ligne n^o2 et de colonne n^o9 du tableau externe.

Plus généralement, une valeur d'adresse est un élément de $M_{adr} = (\mathbf{Z}^+ \cup \mathbf{C})^*$, avec $\mathbf{Z}^+ = \bigcup_{n>0} \mathbf{Z}^n$.

Notons que le '+' dans '$\mathbf{Z}^+$' représente l'ensemble des dimensions de vecteurs, alors que l'*'étoile* symbolise la concaténation des mots d'adresse.

2.1.3 Une structure de monoïde

2.1.4 Déplacement dans une structure

Le déplacement dans une structure de données ou l'extension de celle-ci par ajout d'un étage (le plus interne) suivent une règle similaire et peuvent être représentés par une unique opération de monoïde.

La translation dans un tableau, vers la tête comme vers la queue, peut être représentée par l'addition, notée '+', dans $\mathbf{Z}$ pour un tableau de dimension 1, dans $\mathbf{Z}^n$ pour un tableau de dimension n.

En considérant toujours TAT, un déplacement au niveau d'un tableau est par exemple le passage de la référence $\binom{2}{9}$ vers $\binom{2}{3}$ par l'opération:

$$\binom{2}{9} + \binom{0}{-6} = \binom{2}{3}$$

Le déplacement dans un arbre travaille dans le *monoïde libre* non commutatif pour l'opération de concaténation, généré par l'alphabet des labels de direction (par exemple l et r pour un arbre binaire). Par exemple, le passage de la référence $\binom{2}{9}rl$ vers $\binom{2}{9}rllr$:

$$\binom{2}{9}rl \cdot lr = \binom{2}{9}rllr$$

Cette opération présente une différence avec l'addition dans $\mathbf{Z}$. Elle ne permet le parcours dans un arbre que de la racine vers les feuilles. Par exemple, on ne peut remonter du nœud $llrr$ d'un arbre binaire vers le nœud père llr par concaténation d'un autre mot. Cela revient à constater que le monoïde sur les arbres n'est pas un groupe.

On prendra garde que la concaténation des adresses de 2 pointeurs n'a pas de sens; cela ne correspond en aucune façon à un adressage indirect.

Le changement d'étage consiste en la concaténation à l'adresse courante de l'adresse dans *l'étage* abordé. Par exemple: déplacement de $\binom{2}{9}rl$ à $\binom{2}{9}rl3$:

$$\binom{2}{9}rl \cdot 3 = \binom{2}{9}rl3$$

Monoïde d'adresses

Nous définissons finalement M_{adr} comme un monoïde par l'opération de concaténation, que nous noterons '$\bullet$', dont l'interprétation sémantique est sans ambiguïté, et fournie par l'un des 3 cas précédents.

Dans l'étude qui suit, nous supposons que les monoïdes introduits sont du type analysé dans cette section. L'ensemble des adresses d'une structure de données sera un sous-ensemble de l'un de ces monoïdes.

2.2 Les Variables d'induction

Notre but est maintenant d'analyser l'action d'un programme sur ses *variables d'induction.* Rappelons les contraintes du modèle, signalées dans la Section 1.2.3, et explicitées par la Définition qui suit et le type d'opérations licites sur les *variables d'induction.*

- les *variables d'induction* sont dépositaires de l'adressage-mémoire au sein des structures de données
- les *variables d'induction* ne sont modifiables qu'en respect de la **Règle des blocs**
- les opérations sur les *variables d'induction* sont définies dans la Section 2.2.2.

2.2.1 Définition

Définition 2.1

les variables d'induction *portent en exclusivité l'adressage-mémoire: une référence de tableau, d'arbre ou d'une structure de données mixte emboîtant tableaux et arbres est une expression formée sur ces variables.*

Une variable d'induction *dénote soit:*

- *un argument de type* entier *d'une procédure, initialisé à chaque appel par une constante ou par la somme d'une* variable d'induction *et d'une constante*
- *un argument de type* pointeur *d'une procédure, initialisé à chaque appel par une constante ou une* variable d'induction *de type* pointeur *éventuellement déréférencée*
- *une variable de type* entier *ou* pointeur*, initialisée conformément aux 2 items ci-dessus par un bloc particulier dénoté "Bloc" dans notre syntaxe.*
- *un compteur de boucle de type* entier*, translaté d'une constante à chaque itération*
- *un compteur de boucle de type* pointeur*, translaté d'une constante à chaque itération*

Interprétation

Deux notions-clef sont implantées dans cette définition.

- Selon cette définition, une *variable d'induction* ne peut être modifiée que par l'ouverture d'un *bloc* de la syntaxe (cf Section 2.3). Ces ouvertures de *bloc* sont les éléments constitutifs des *instances* (cf Chapitre 4). De cette manière, *l'instance*, abstraction de multiples exécutions possibles, se verra attribuer une *liaison* unique et parfaitement définie de ses *variables d'induction*, c.-à-d. de l'adressage.
- Nous verrons dans la Deuxième partie que les modalités d'initialisation des *variables d'induction* sont modélisables dans le cadre de la théorie des monoïdes, des langages réguliers et des automates. la Section qui suit approfondit l'analyse des modifications des *variables d'induction.*

2.2.2 Opérations sur les variables d'induction

La Définition 2.1 prend en charge les manipulations suivantes sur les *variables d'induction*, comme on peut aisément le vérifier.

Opérations de base

- ré-initialisation à zéro: *remise à zéro*
- incrémentation/décrémentation: *translation*
- copie de la valeur d'une variable d'induction par une autre: *capture*

Ces opérations sont des cas particuliers *d'opération complexe* présentée ci-dessous.

Opérations complexes

- *remise à zéro* suivie d'une *translation*: *initialisation*
- *capture* suivie d'une *translation*: *écrasement*
- *mixture*: c'est l'union, pour plusieurs ou pour l'ensemble des *variables d'induction*, d'opérations simples ou complexes.

  Par exemple, une instruction peut commander la *permutation* des valeurs d'un groupe de *variables d'induction.* Si ce groupe est un couple, il s'agit d'un *échange.*

Remarque

Toute séquence composée d'opérations des types précédents peut être fusionnée en une seule *mixture.*

Type_prédéfini	==	*type_entier* | *type_flottant* | *type_caractère* | *type_chaîne* | *type_booléen*
Opérateur_prédéfini	==	... opérateurs habituels
Type	≙	*Type_adresse* | *Type_donnée* | *Type_structuré*
Type_adresse	≙	*type_entier* | *type_chaîne* | *Type_adresse**
Type_donnée	≙	*Type_prédéfini* | *Type_enregistrement*
Type_enregistrement	≙	(id_champ: *type_chaîne*, champ: *Type_donnée*)*
Type_structuré	≙	*Type_Tableau* | *Type_Arbre*
Type_Tableau	≙	id_tab: *type_chaîne*, dim: *type_entier*, *Type*
Type_Arbre	≙	id_arbre: *type_chaîne*, directions: *type_chaîne*, *Type*
Expression	≙	*Constante* | *Variable* | *Binaire* | *Unaire*
Constante	≙	id_const: *type_chaîne*, donnée: *Type_donnée*
Variable	≙	*Variable_déclarée* | *Elément* | *Noeud*
Variable_déclarée	≙	id_var: *type_chaîne*, *Type*, valeur: *Expression*
Elément	≙	tableau: *Type_Tableau*, indice: *type_entier*
Noeud	≙	arbre: *Type_Arbre*, mot: *type_chaîne*
Binaire	≙	terme_1,terme_2: *Expression*, op_bin: *Opérateur_prédéfini*
Unaire	≙	op_un: *Opérateur_prédéfini*, terme: *Expression*
Programme	≙	partie_déclarative: *Procédure*$*$, main: *Procédure*
Procédure	≙	id_proc: *type_chaîne*, paramètres: *Variable_déclarée**, corps_proc: *Instruction*
Instruction	≙	*Bloc_Conditionnel* | *Boucle_for* | *Bloc* | *Instruction_terminale* | *Instruction**
Instruction_terminale	≙	**nulle** | **affectation** | **appel**(id_proc: *type_chaîne*)↓
Bloc_Conditionnel	≙	**test_if**, then: *Instruction*, else: *Instruction*
Boucle_for	≙	**initialisation**, corps_bouc: (**test_in**, *Instruction*), **itération_plus**↓
Bloc	≙	déclaration: *Variable_déclarée**, corps_bloc: *Instruction*

Figure 1. La syntaxe des programmes

2.3 La syntaxe des programmes

2.3.1 Une syntaxe *ad hoc*

Cette syntaxe a été élaborée en tenant compte de plusieurs impératifs relatifs à son exploitation:

- mettre l'accent sur le flot de contrôle, c.-à-d. sur la formation des *traces d'exécution*;
- construire *l'automate de contrôle* du programme;
- visualiser et donc réaliser simplement la *procédure d'oubli*, qui permet le passage du *mot de trace* au *mot de contrôle*;
- permettre le suivi précis des opérations sur les *variables d'induction* et relayer ce suivi dans les *mots de contrôle*.

La définition de la syntaxe apparaît Figure 1. Nous l'avons décomposée en 2 parties.

2.3.2 Syntaxe des structures de données et des expressions

La *SSDE* est conforme aux explications fournies en Section 3.1.
La notation ' * ' est l'opérateur de liste des expressions régulières: ici, il représente la *répétition séquentielle*.

2.3.3 Syntaxe des instructions

L'analyse de cette syntaxe en perspective avec la sémantique que nous visons sera détaillée après l'introduction des *variables d'induction* et des opérations associées. Nous ne brossons ici qu'une vue d'ensemble.
Hors la ponctuation, la syntaxe est construite à l'aide de 2 types d'entités appelées *constructions*:

- les *Non Terminaux*, qui apparaissent en partie gauche des règles, sont écrits *en Italiques*, et ont une majuscule de tête; les *Non Terminaux* composants d'agrégat sont précédés d'un *terminal valué* en style minuscule suivi de ' : ';
- les **terminaux**, écrits en minuscules **gras** non italiques; il existe 2 **terminaux** particuliers, terminés par ' ↓ '; on en verra la signification plus loin.

Le symbole ' , ' est un constructeur d'*agrégat*.
Le symbole ' * ' dénote une liste de cardinal quelconque.
Le *Non Terminal Instruction* joue le rôle du *bloc* présenté en Section 2.3.

2.4 Sémantique des programmes

Nous ne mentionnons que les informations utiles pour clarifier la compréhension.

- **affectation**: il s'agit d'une affectation quelconque à condition qu'elle ne modifie aucune des *variables d'induction*; plus généralement, on peut étendre la liste des *Instruction_terminale* à toute instruction élémentaire utilisée par un langage;
- **appel**(id_proc:*type_chaîne*)↓: appel récursif ou non récursif de procédure. Le symbole '↓' signifie que l'instruction renvoie sémantiquement au corps de la procédure passée en paramètre;
- *Boucle_for*: l'initialisation de boucle, **initialisation**, porte sur une *variable d'induction*; le test de boucle, **test_in**, est à l'entrée du corps de boucle;
- **itération_plus**↓: identifie les itérations ultérieures de la boucle; cette instruction est supposée débuter avec l'incrémentation de l'indice (*variable d'induction*); Le symbole '↓' signifie qu'une fois l'incrémentation effectuée, l'instruction renvoie sémantiquement au corps de boucle;
- *Bloc*: Cette instruction initialise une ou plusieurs *variables d'induction*;

- *Bloc_Conditionnel*: la sémantique de la '**,**' est évidemment qu'une exécution s'oriente en exclusivité vers l'une ou l'autre des 2 branches;
  tout *Bloc_Conditionnel* est formé de ses 2 branches, la branche ***else*** abritant éventuellement l'instruction **nulle**;

- **nulle**: instruction sans effet; on ajoutera cette *Instruction* en fin du corps de *Programme* si cela est nécessaire pour assurer la présence d'une unique instruction de fin de programme, que nous appelons **the_end**. Cela simplifie les explications.

Rappelons l'exigence de la **Règle des blocs** annoncée au Chapitre 2. Ici, le *bloc* est l'*Instruction*.

2.5 La Syntaxe abstraite des procédures

2.5.1 Motivations

Notre *syntaxe abstraite des procédures* (*SAP*) a pour finalité d'établir un pont entre les *traces* du programme et les *mots de contrôle* associés. Ce pont sera *l'automate de contrôle*. Elle est montrée Figure 2
Une procédure est définie par l'arbre construit selon la *SAP*. Les nœuds de l'arbre correspondent aux ***terminaux*** *valués* qui qualifient les *Non_terminaux* et aux **terminaux**.

L'arbre de dérivation d'une procédure s'appellera *arbre de procédure*, celui de la procédure principale *arbre du programme*, et l'ensemble des *arbres de procédure* d'un programme sa *forêt*. Un programme sera donc défini en général non par un seul arbre, mais par une *forêt* de syntaxe abstraite.
Par analogie, nous définirons l'*arbre de boucle* comme l'arbre de dérivation sous la racine ***bloc_for***.

La *SAP* nous fournira les briques de base pour construire les *traces* du programme p d'une part, *l'automate de contrôle* d'autre part. Ces briques sont les *arbres* de toutes les procédures de p.

La *SAP* se débarrasse des détails inutiles eu égard à ce projet: il s'agit d'établir une **correspondance** entre les sous-arbres de l'arbre de syntaxe abstraite de la procédure et les *blocs*, identifiés par le Non_terminal *Instruction*.

Procédure	$\triangleq$	***id_proc***: *Instruction*
Instruction	$\triangleq$	*Bloc_Conditionnel* | *Boucle_for* | *Bloc* | *I_terminale* | *Instruction**
I_terminale	$\triangleq$	**nulle** | **affectation** | **appel(*id_proc*)**$\downarrow$
Bloc_Conditionnel	$\triangleq$	***bloc_if***: (***then***: *Instruction* ***else***: *Instruction*)
Boucle_for	$\triangleq$	***bloc_for***: (*Instruction* **itération_plus**$\downarrow$)
Bloc	$\triangleq$	***bloc***: *Instruction*

Figure 2. La syntaxe abstraite des Procédures

2.5.2 Comparaison avec la syntaxe des Programmes

La *SAP* se situe au niveau des instructions: elle fait abstraction de la *SSDE*.

Elle n'explicite plus les déclarations ni les types. Les **terminaux** et les ***terminaux*** *valués*, repérés respectivement par les style **gras** et ***italiques gras***, sont les nœuds utiles de *l'arbre de syntaxe abstraite* (*ASA*). Ce sont ces nœuds que nous repérons à l'aide d'un **label**.

Il y a un seul *Non_terminal* en partie gauche, -*Instruction*-, non référencé par un ***terminal*** *valué*: ce dernier ferait double emploi avec celui qui définit cette *Instruction* en partie droite d'une règle.

Domaine de l'analyse

Notre sémantique prend en compte l'ensemble des exécutions possibles *abstraction faite des choix opérés tant lors des tests conditionnels que des tests de boucle.* Il s'agit donc d'une approximation conservatrice, c'est-à-

dire qui envisage des exécutions possiblement non réalisables. Cette licence, outre qu'elle est moins gourmande en informations, permettra par la suite de placer les calculs dans le domaine des langages rationnels et des automates finis associés.

Les tests sont donc ignorées de notre SAP.

Syntaxe de la boucle

L'**initialisation** d'une boucle est confondue avec l'entrée dans cette boucle sous le ***terminal*** *valué* ***bloc_for***. Ce dernier cumule également le rôle d'indicateur pour l'entrée dans le corps de boucle, c.-à-d. dans la $1^{\text{ère}}$ itération.

2.5.3 Le programme Queens

Nous illustrons notre propos à l'aide de l'exemple (emprunté à [Coh99]) de la Figure 3. Ce programme calcule toutes les solutions du *Problème des n Reines*. On notera que nous avons omis de labéliser les instructions ***bloc_if***, ainsi que les branches ***else*** (d'ailleurs absentes du texte source en C) par souci d'allègement: cela n'entravera pas la précision de l'analyse.

La Figure 3.a est le masque de la SAP formé des *noms des constructions* auxquelles s'unifient les instructions du programme.

La Figure 3.b est le résultat de cette unification, c'est la *forêt* de la syntaxe abstraite du programme.

La Figure 3.c est le pendant de la Figure 3.b en termes de **labels**.

```
                                        int A[n];
X    id_proc                             void Queens(int n, int k) {
                                           if (k < n)
T    then                                     {
F    bloc_for   /  f  itération_plus↓              for (int i = 0; i < n; i++) {
G    bloc_for   /  g  itération_plus↓                 for (int j = 0; j < n; j++)
r    affectation                                         ... = ... A[j] ...;
                                                  if (...)
U    then                                           {
s    affectation                                       A[k] = ...;
Q    appel(id_proc)↓                                    Queens (n, k+1);
                                                     }
                                                }
                                             }
                                        }

S    id_proc                             int main() {
R    appel(id_proc)↓                           Queens (n,0);
                                        }
```

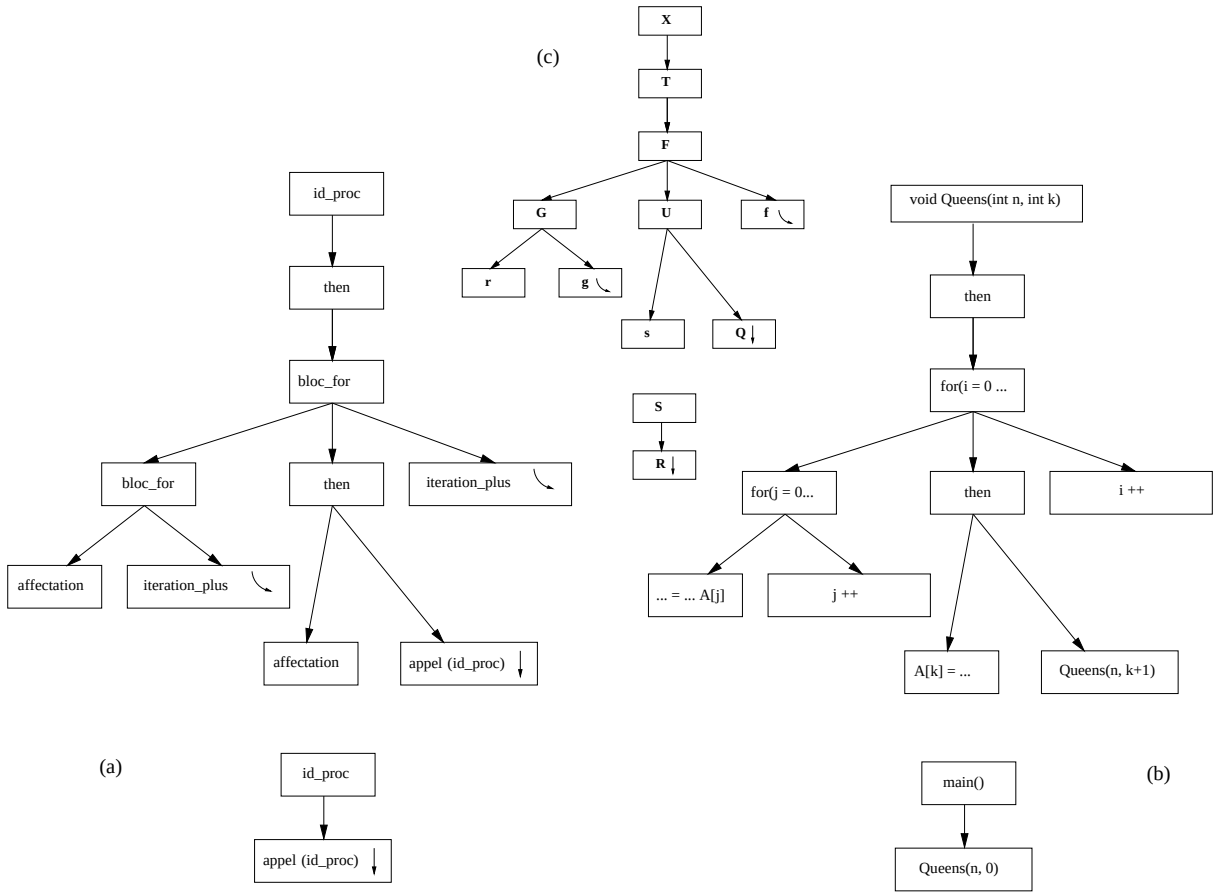

Figure 3. Le programme **Queens**

Chapitre 3

La génération des traces d'un programme

3.1 La génération des traces

3.1.1 Labels et occurrences

Une exécution étant donnée, toute *occurrence* o du programme est identifiable par la suite des **labels** associés chacun à une *occurrence* exécutée, depuis le début du programme jusqu'à o compris. Cette suite, le *mot de trace*, a pour dernier **label** celui de l'instruction dont o est une occurrence. Si cette instruction n'est pas **the_end**, la trace est dite *partielle*. Nous allons définir formellement une trace comme un *arbre*.

3.1.2 Algorithme de génération des traces

Il construit l'*arbre d'une trace* à partir de *la forêt du programme.*

Algorithme 1

- 1. Considérer l'*arbre du programme* ;
- 2. Pour chaque *bloc_if*: choisir une des branches, éliminer l'autre;
- 3. Pour chaque *bloc_for*: si la boucle n'exécute aucune itération, supprimer le sous-arbre de racine *bloc_for*;
- 4. Pour tous les terminaux *fléchés*, Faire:
  - 4.1. si c'est un terminal *fléché* appel(*id_proc*)↓:
    supprimer ce terminal;
    "greffer" à la place *l'arbre* A de la procédure *id_proc*;
    substituer au nœud *id_proc* racine de A le nœud appel(*id_proc*)↓;
  - 4.2. si c'est un terminal *fléché* itération_plus↓, noté t :
    supprimer ce terminal t;
    si la boucle ne comporte plus d'itération: ne rien faire;
    si la boucle ne comporte des itérations supplémentaires:
    "greffer" à la place *l'arbre de la boucle*, noté A;
    remplacer le nœud *bloc_for* racine de A par t;

  Fait;
  retourner en 2.
- 5. Fin

Proposition 1
L'Algorithme 1 génère l'un quelconque des arbres de trace *d'un programme.*

Justification:
Cette proposition n'est pas soumise à une procédure de preuve, car nous n'avons jusque là pas proposé de définition pour l'exécution d'un programme. En fait, elle peut être vue comme la sémantique opérationnelle attribuée à notre syntaxe.

Mot de trace

L'Algorithme enregistre chacune des *occurrences* exécutées sous la forme d'un nœud. Soit o une *occurrence* et n le nœud associé. Le parcours préfixe en profondeur à main gauche ($PPPMG$) de l'arbre depuis la racine jusqu'à n produit par concaténation des *labels* des nœuds rencontrés un mot distinctif de l'*occurrence* o, le *mot de trace*.

On dira que *l'arbre de trace* et le *mot de trace* sont *dérivés* ou sont des *dérivations* de la procédure (ou du programme).

3.1.3 Une trace de Queens

Nous présentons en Figure 4 *l'arbre* qui représente une *trace* de **Queens**.

Le *mot de trace* associé est :

SRTFGrgrUsQTFGrUsfUsQTFUs

3.2 La terminaison d'un programme

3.2.1 Notion de terminaison

Nous restreignons notre propos à des programmes sans code mort: toutes les procédures peuvent être appelées depuis la procédure principale. De plus, nous ne considérons que les exécutions qui terminent, c.-à-d. des *traces complètes*.

Une *trace complète* est une trace finie, construite par l'Algorithme 1, dont la dernière instruction est **the_end**.

Nous dirons d'un programme ou d'une procédure qu'ils *terminent* si la génération *partielle* d'une trace quelconque suivant l'Algorithme 1 peut **toujours** être *complétée*.

Celui-ci donne une caractérisation des traces qui peuvent être *complétées*. Une exécution *partielle* de cet Algorithme produit un *arbre de trace incomplet* A. Un tel arbre a les caractéristiques suivantes:

- il contient des **terminaux** *fléchés*;
- il contient possiblement les 2 branches d'un test conditionnel (*alternative*).

Compléter la trace consiste à poursuivre l'exécution de l'Algorithme jusqu'a l'étape 5. On obtient alors un arbre qui ne contient plus ni **terminaux** *fléchés* ni *alternative*. Nous dirons que l'arbre A est *terminable*. Si A est l'*arbre d'une procédure* p, nous dirons de même que p est *terminable*.

Remarque:
Le terme *terminable* est plus faible que *termine*: une procédure *termine* si **tout** *arbre de trace incomplet* qui en dérive est *terminable*.

3.2.2 Chaîne d'appels mutuellement récursifs

Définition 3.1

Une chaîne d'appels *de procédures est une suite d'appels tel que chacun soit inclus dans le corps de procédure de l'appel précédent.*

Une chaîne d'appels mutuellement récursifs *(CAR) de procédures est une* chaîne d'appels *qui est:*
- fermée: les premier et dernier appels concernent une même procédure;
- élémentaire: elle ne contient pas de sous-chaîne fermée.

Une CAR est repérable sur *l'arbre de trace* par une portion de chemin dont le mot associé a pour premier et dernier **labels** 2 **labels** de la même procédure, et des **labels** intérieurs sans référence à cette dernière. Nous appellerons 'CHR' une telle portion de chemin récursif, qui est de la forme '(**appel**($\boldsymbol{P}$)...**appel**($\boldsymbol{P}$))'.

3.2.3 Règle de terminaison

Lemme 3.1

Un programme termine *si et seulement si toutes les procédures du programme, dont la procédure principale, sont* terminables.

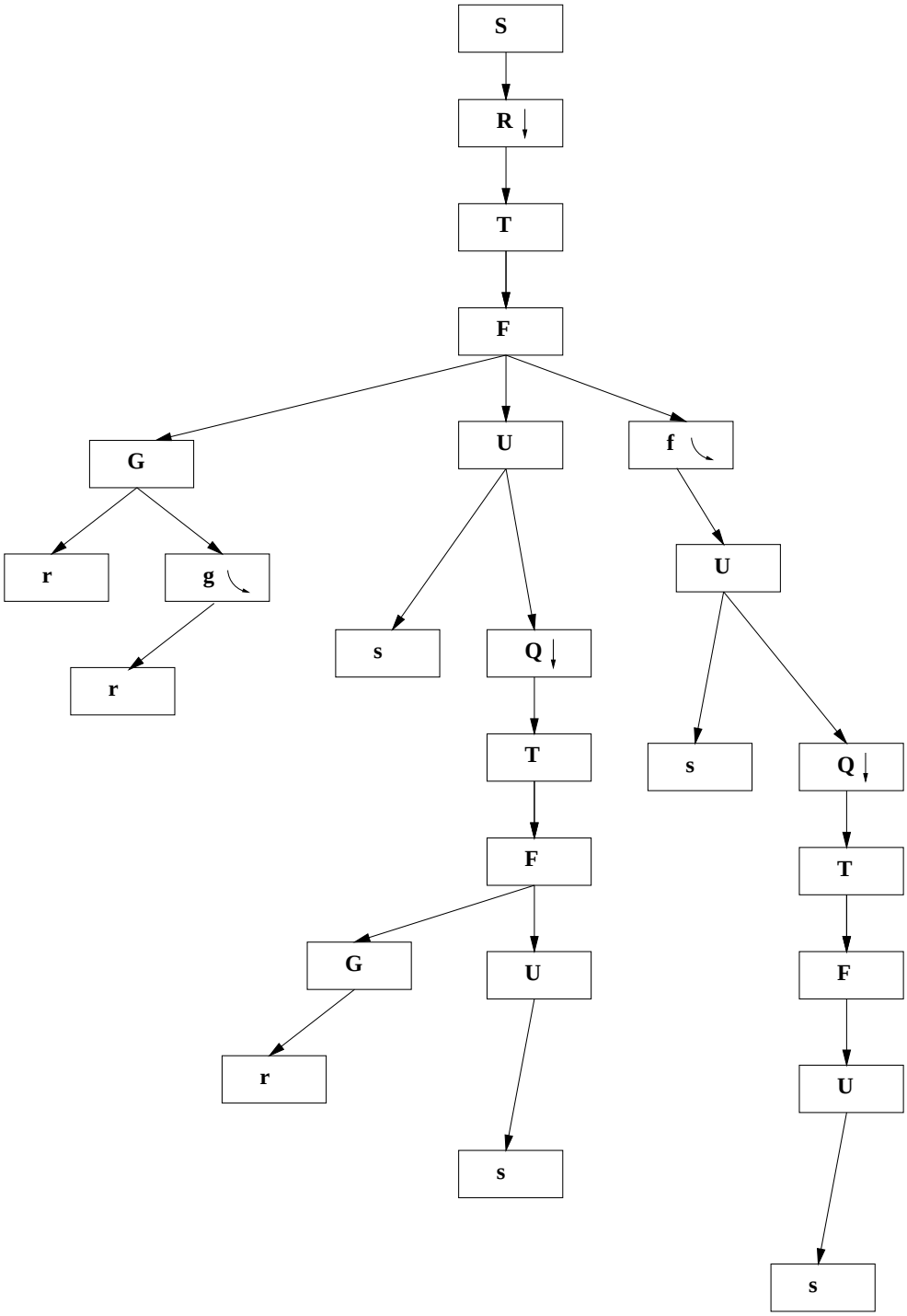

Figure 4. Un arbre de trace du programme **Queens**.

Preuve:
Soit un *arbre de trace incomplet* A. Pour compléter A, il suffit de supprimer les **terminaux** *fléchés* de boucle et de substituer une dérivation *complète* à chaque flèche de procédure. Réciproquement, toutes les procédures du programme sont concernées: l'Algorithme 1 permet de construire A en y laissant figurer une procédure quelconque du programme (pas de code mort). □

Lemme 3.2
Une procédure p est terminable *si et seulement si il existe un* arbre de trace *de p qui ne contient pas de CHR.*

Preuve:
Supposons *p terminable* et A un *arbre de trace* de p. Montrons comment éliminer les CHR introduites dans A.
Soit c une CHR de A, dont les premier et dernier appels sont les nœuds N_1 et N_2. Substituons dans A le sous-arbre de racine N_2 au sous-arbre de racine N_1. Le nouvel arbre est légal et ne contient plus c. Nous pouvons renouveler cette opération jusqu'à épuisement de toutes les CHR de A.
Réciproquement, s'il existe un *arbre de trace*, par définition p est *terminable*. □

Critère caractéristique d'une procédure terminable

L'analyse de *l'arbre du programme P* va nous permettre de décider si celui-ci *termine*. L'idée est d'y repérer les CHR potentielles de toutes les traces possibles et de s'assurer que pour chaque procédure, qui sont toutes présentes puisqu'il n'y a pas de code mort, il existe une dérivation qui les évite. Pour cela, nous construisons un *arbre logique* du programme.

Algorithme 2: construction de l'arbre logique du programme

Soit A l'arbre du programme P.

- 1. Supprimer les terminaux itération_plus↓.
- 2. Tant_que: il existe un terminal appel(id_proc)↓ Faire :
  si ce n'est pas le dernier appel(id_proc)↓ d'une CHR:
  supprimer ce terminal;
  "greffer" à la place *l'arbre* de la procédure *id_proc*;
  supprimer les terminaux itération_plus↓ ;
  Fait.
- 3. Attribuer un *label booléen* aux feuilles:
  FALSE pour appel(id_proc)↓,
  TRUE pour les autres terminaux.
- 4. Calculer le *label booléen* des autres nœuds:
  un nœud *bloc_if* vaut pour un *OU* logique de disjonction de ses fils,
  un autre nœud (nœud d'instructions en séquence) vaut pour un *ET*
  logique de conjonction de ses fils.

Les sous-arbres de l'*arbre logique* dont la racine est un **label** de procédure (nœud ***id_proc***) sont appelés les *arbres logiques de procédure*. Toutes les procédures sont représentées par un exemplaire au moins de leur *arbre logique*.

Lemme 3.3
Un programme termine *si et seulement si la racine de tous les* arbres logiques de procédure *ont pour valeur* ***TRUE***.

Preuve:
Par construction, l'*arbre logique* est un *arbre de trace incomplet*. Sa racine est le **label** de la procédure principale.
Supposons que la racine de l'*arbre logique*, noté L, est **TRUE**. Supprimons, parmi les branches ***then*** et ***else*** des ***bloc_if***, celles dont la racine est étiquetée **FALSE**. Ces branches sont à exclure d'une trace valide respectant le Lemme 3.2. Supprimons au choix l'une quelconque des 2 branches de chaque autre ***bloc_if***. Le nouvel arbre, qui ne contient plus de nœud **FALSE**, est un *arbre de trace*.

Inversement, Supposons que la racine de L est **FALSE**. Le choix d'une branche de chaque ***bloc_if*** ne change pas la valeur logique de la racine de L. Cela implique qu'on ne peut construire une trace sans nœud **FALSE**: tout *arbre de trace* en construction par l'Algorithme 1 contiendra une CHR. Le lemme 3.2 permet de conclure: P ne *termine* pas. □

Critère caractéristique d'un programme qui termine

Pour éviter de calculer plusieurs fois la valeur logique des sous-arbres de procédure, il suffit de transformer L_p en un graphe orienté acyclique (*Directed Acyclic Graph ou DAG*) qui ne référence qu'une seule fois chaque procédure, le *DAG logique* du programme. Pour construire ce dernier, il suffit de modifier l'étape 2 de l'Algorithme 2 pour fusionner les sous-arbres de la même procédure. L'étape 2 est réécrite comme suit:

```
- 2. Tant_que: il existe un terminal appel(id_proc)↓ Faire :
      si ce n'est pas le dernier appel(id_proc)↓ d'une CHR:
         si l'arbre a de la procédure n'est pas déjà dans le DAG :
           supprimer ce terminal;
           "greffer" à la place l'arbre a de la procédure id_proc;
           supprimer les terminaux itération_plus↓;
         sinon:
           construire un arc depuis le père du terminal vers la racine
           de a ;
      Fait.
```

Le test de terminaison d'un programme se redéfinit finalement comme suit:

Théorème 3.1
Un programme termine *si et seulement si l'entrée et tous les nœuds internes* ***id_proc*** *de son* DAG logique *sont étiquetés* ***TRUE***.

3.2.4 Travaux futurs

La modélisation des graphes de grammaire 'ET-OU' introduite par B. Lang [Lan91] pourrait être un outil adapté pour une reformulation du critère de terminaison. En effet, ce dernier concept semble jumeau de celui de *grammaire réduite* tel que présenté dans [Aut94] et rappelé en Annexe. Cela suppose cependant la définition d'une grammaire des traces. Une telle redéfinition des traces alternative à un algorithme de ré-écriture d'arbres, toujours à partir de la *syntaxe abstraite des procédures*, ne semble pas présenter de difficultés particulières. Cela souligne simplement que ré-écriture d'arbres et dérivation de grammaire sont des notions équivalentes.

Chapitre 4

Les mots de contrôle

4.1 Abstraction des traces

4.1.1 Mot sur un chemin d'arbre

Pour les raisons explicitées au Chapitre 2, nous allons travailler non pas sur les traces *in extenso* mais sur une *abstraction* des traces.

Considérons un *arbre de trace* A et une *occurrence* o de cette trace. A o est associé un *mot de trace*, dont le dernier **label** est l'étiquette d'un nœud sur A. Le chemin sur l'arbre depuis la racine jusqu'a o est identifié par le mot formé des **labels** des nœuds rencontrés.

Définition 4.1
Soit o une occurrence *sur un* arbre de trace A. *Le* mot de contrôle *de o est le* mot du chemin *sur A depuis la racine jusqu'à o.*

4.1.2 La relation d'oubli

Il est aisé de retrouver le *mot de contrôle* à partir du *mot de trace*. Nous *simplifions* le *mot de trace* par la procédure suivante que nous baptisons *oubli*:

Reformulation 4.1
Considérons le parcours PPPMG qui génère sur l'arbre de trace *le* mot de trace *de l'*occurrence o.
Dans le mot de trace, *nous rayons tout sous-mot qui est témoin du parcours PPPMG complet d'un sous-arbre de A dont la racine n'est pas un ancêtre de o. Le mot résultant est le* mot de contrôle.

Considérée du point de vue plus concret de l'exécution, cette seconde définition peut être reformulée:

Reformulation 4.2
Dans le mot de trace, *nous rayons tous les sous-mots témoins (de l'exécution complète) d'un* bloc d'instructions refermé. *Seuls subsistent les* **labels** *des* blocs d'instructions ouverts et non refermés. *Le mot résultant est le* mot de contrôle.

La fonction d'oubli
Cette procédure définit une *fonction d'oubli* Φ de Σ^*_{ctrl} dans L_{ctrl} qui transforme un *mot de trace* en *mot de contrôle*.

Nous utiliserons ci-dessous la fonction W_e de O_e dans L_{ctrl} qui fournit le *mot de contrôle* associé à une *occurrence* d'une exécution e du programme P.

4.1.3 L'instance

Du *mot de trace* au *mot de contrôle*
Si nous considérons une exécution particulière e du programme définie par *l'arbre de trace t*, une *occurrence* o de e détermine un *mot de trace* et un *mot de contrôle*, tous deux uniques. Le *mot de contrôle* est donné par $W_e(o)$. La connaissance de e ou plus directement de t d'une part, et du *mot de trace* d'autre part, permet de trouver le *mot de contrôle*.

Du *mot de contrôle* au *mot de trace*

Inversement, la connaissance de *l'arbre de trace* t et du *mot de contrôle* va redonner le *mot de trace*, ce qu'exprime le Théorème suivant:

Théorème 4.1

La fonction W_e est injective.

Preuve:

Remarquons en premier lieu que la connaissance de t et du *chemin* associé à o sur t redonne le *mot de trace*. Il suffit alors de montrer que le *mot de contrôle* permet de trouver un et un seul *chemin* sur t.

Nous dotons l'ensemble des **labels** du programme d'une relation d'ordre partiel noté $<_{lab}$:
Soit l_1 et l_2 2 **labels**, $l_1 <_{lab} l_2 \Leftrightarrow$ l_1 est un frère aîné (gauche) de l_2 sur l'*arbre d'une procédure* du programme. Par construction de *l'arbre de trace* t, si l_1 est un frère aîné de l_2 sur t, alors $l_1 <_{lab} l_2$. Par conséquent, $<_{lab}$ génère un *ordre lexicographique* $<_{lex}$ sur les *mots de chemin* de t isomorphe à *l'ordre lexicographique* $<_{lex}$ sur les *chemins* de t. □

L'oubli

Considérons l'ensemble E des exécutions finies du programme, obtenues en parcourant toutes les combinaisons possibles de résultats des tests conditionnels et de boucle. Soit O_e l'ensemble des *occurrences* de l'exécution e, et O l'Union des O_e pour toutes les exécutions e de E.

En toute généralité, un *mot de contrôle* représente maintenant un ensemble, infini (si le programme contient des boucles ou des cycles d'appel récursifs), ou fini de *mots de trace* de O.

On peut alors définir une *relation d'oubli* dans l'ensemble O. Deux *occurrences* sont en relation si et seulement si elles produisent le même *mot de contrôle*. C'est trivialement une relation d'équivalence. L'ensemble-quotient qui en résulte est un ensemble de *classes d'occurrences*, chacune dénotée par le *mot de contrôle* associé.

Définition 4.2

Une instance *est une* classe d'occurrences *modulo la relation* d'oubli *sur les mots. C'est l'ensemble de toutes les* occurrences *dont le* mot de trace *produit par* oubli *le même* mot de contrôle.

Cette définition établit un isomorphisme entre les *instances* et le langage des *mots de contrôle*.

Une instance pour plusieurs occurrences

Soit o une *occurrence* dont l'historique relate l'exécution d'une boucle, antérieure à o, composée de n itérations et soit o' l'*occurrence* identique à o à l'exception de la boucle, composée cette fois de n' itérations, avec $n' \neq n$. Les *mots de contrôle* associés respectivement à ces 2 *occurrences* sont identiques.

Si l'historique de o relate cette fois-ci l'exécution de la branche ***then*** d'un *Bloc_Conditionnel* refermé, tandis que o' relate l'exécution de la branche ***else***, les *mots de contrôle* associés sont également identiques.

Enfin, il en est de même si les 2 *occurrences* sont identiques abstraction faite de l'exécution consommée d'un appel de procédure, exécution différente d'une *occurrence* à l'autre.

Dans chacun de ces exemples, une *instance* représente plusieurs *occurrences* de O.

4.1.4 Une instruction: 3 niveaux sémantiques

3 niveaux de signification d'une instruction apparaissent dans le texte source:

- *l'instruction-source*, délimitée par 2 *points de programme*; son **label** ferme tous les *mots de trace* qui s'achèvent avec l'exécution de cette instruction.

  Soit s une instruction interne au corps d'une boucle. Soit o et o' deux *occurrences* de s appartenant à 2 itérations distinctes de la boucle. o et o' sont les *occurrences* de 2 *instances* distinctes.

  Il en sera de même si maintenant s est une instruction interne au corps d'une procédure p, et si o et o' sont deux *occurrences* de s produites respectivement par l'activation de 2 appels distincts de p.

- *l'instance* identifie une unique *occurrence* dans une exécution, mais plusieurs dans l'ensemble des exécutions possibles.

- *l'occurrence* est le niveau concret de l'exécution.

4.2 L'automate de contrôle

4.2.1 Construction de l'automate

Définition 4.3

L'automate de contrôle $\mathcal{A}$ *d'un programme* P *est un* automate fini *dont* ***les états sont les nœuds de*** **la forêt du programme**, *nœuds* **appel*(id_proc)*↓** *et* **itération_plus↓**) *exclus. L'état de la racine de* l'arbre, *noté START, représente l'appel de la procédure principale.* ***Tous les états sont d'acceptation (finaux).*** ***La définition des transitions*** *est réalisée comme suit:*

- *A chaque nœud* **appel(*id_proc*)↓** n_a *de la* forêt *correspond une transition (un arc) de l'automate reliant le père de a au nœud* ***id_proc****, racine de la procédure appelée. L'étiquette de la transition est le* **label** *du nœud* n_a.
- *A chaque nœud* **itération_plus↓** n_i *de la* forêt *correspond une transition (un arc) de l'automate établissant une boucle du père de* n_i*, qui est un nœud* ***bloc_for****, sur lui-même. L'étiquette de la transition est le* **label** *du nœud* n_i.
- *Finalement, à chaque arc de la* forêt *correspond une transition de l'automate reliant les états correspondants. L'étiquette de la transition est le* **label** *du nœud-fils de la* forêt.

L'automate de Queens

L'automate associé au programme **Queens** est présenté Figure 5 en regard du texte source.

```
          int A[n];
X         void Queens(int n, int k) {
              if (k < n)
T                 {
F/f                 for (int i = 0; i < n; i++) {
G/g                     for (int j = 0; j < n; j++)
r                           ... = ... A[j] ... ;
                        if (...)
U                           {
s                             A[k] = ... ;
Q                             Queens (n, k+1);
                            }
                    }
                }
          }

S         int main() {
R                 Queens (n,0) ;
          }
```

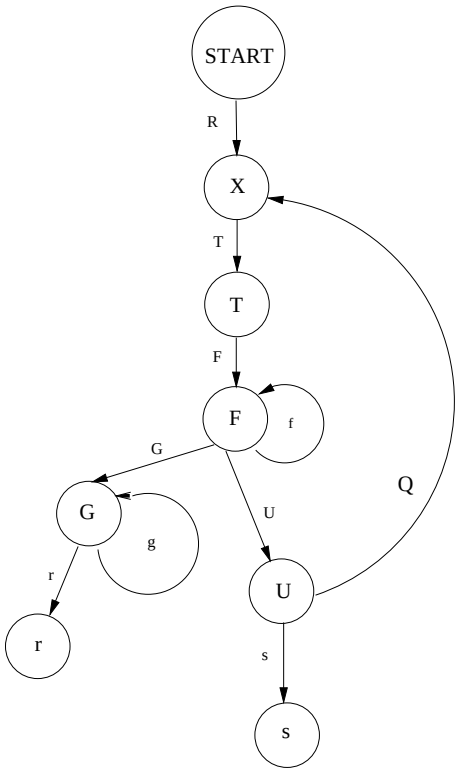

Figure 5 . Le programme **Queens** et son automate de contrôle.

Si nous limitons les états d'acceptation aux états r et s, le langage reconnu par cet automate est :

$$w = \mathrm{RTF(f + UQTF)^*(Us + Gg^*r)} \tag{4.2.1}$$

Nous allons montrer dans le cas général que ce langage est celui des *mots de contrôle.*

4.2.2 Automate et mots de contrôle

L'objectif de cette Section est de montrer que l'automate reconnaît les *mots de contrôle*. Nous commençons par montrer que les chemins sur *l'arbre de trace* se retrouvent sur le graphe associé à l'automate. Pour cela, nous introduisons la Définition suivante:

Définition 4.4
L'arbre générique des traces *(AGT) d'un programme est un* arbre infini rationnel *conforme à l'Algorithme interminable (dans le cas général) qui suit:*

Algorithme 3: génération de *l'arbre générique des traces*

- *1. Construire l'*arbre du programme*;*
- *2. Pour tous les terminaux* fléchés*, Répéter en boucle:*
  - *2.1. si c'est un terminal* fléché *appel(id_proc)↓:*
    supprimer ce terminal;
    "greffer" à la place l'arbre *A de la procédure* id_proc*;*
    substituer au nœud id_proc *racine de A le nœud appel(id_proc)↓;*
  - *2.2. si c'est un terminal* fléché *itération_plus↓:*
    supprimer ce terminal t;
    "greffer" en lieu et place l'arbre de la boucle*, noté A;*
    remplacer le nœud bloc_for racine de A par t;

Le nombre de **terminaux** *fléchés* étant fini, l'*AGT* est *rationnel* et peut être représenté par un diagramme fini, le *diagramme des traces.*

Remarque 1
De l'AGT, on peut dériver n'importe quel arbre de trace*, comme on le vérifie aisément par comparaison des Algorithmes 1 et 3:* l'arbre de trace *est un arbre issu de l'AGT à la suite d'un élagage particulier.*

La vérification du Lemme ci-dessous est immédiate.

Lemme 4.1
Le diagramme des traces *est identique au graphe de l'automate.*

Lemme 4.2
Soit a et b 2 nœuds sur l'arbre de trace *d'une exécution du programme.*
1. b est fils de a sur l'arbre de trace *si et seulement si b est fils de a sur l'AGT.*
2. b est fils de a sur l'AGT si et seulement si b est successeur immédiat *de a sur le* diagramme des traces.
3. c est un chemin sur l'arbre de trace *si et seulement si c'est un chemin sur l'AGT.*
4. c est un chemin sur l'AGT si et seulement si c'est un chemin sur le diagramme des traces.
5. c est un chemin sur l'arbre de trace *si et seulement si c est un chemin sur le* diagramme des traces.

Preuve:
(1) dérive de la Remarque 1.
(2) exprime une propriété du diagramme d'un arbre rationnel.
(3) est induit par (1).
(4) est induit par (2).
(5) se déduit de (3) et (4) par transitivité. □

Théorème 4.2
Etant donné un programme qui termine*, le langage des* mots de contrôle *est le langage reconnu par* l'automate de contrôle. *C'est un* langage rationnel*, noté L_{ctrl}, inclus dans Σ^*_{ctrl}.*

Preuve:
Le Lemme 4.2.5 assure que tout *mot de contrôle* est reconnu par l'automate.
Réciproque:
Soit un mot m reconnu par l'automate. m n'est un *mot de contrôle* que s'il existe un *arbre de trace* complet dont il est un *mot de chemin.*

m est le mot associé à un chemin c sur l'AGT d'après les Lemmes 4.1 et 4.2.4. Comme ce chemin est fini, il appartient à une construction incomplète, notée AGt, de l'AGT selon l'Algorithme 3: la boucle *Répéter* est exécutée un nombre fini de fois.

AGt est aisément transformable en un *arbre de trace incomplet* qui contient c: il suffit de sélectionner pour chaque *Bloc_Conditionnel* qui détermine c la branche qui porte c, le choix étant indifférent pour les *Bloc_Conditionnel* indépendants de c. Lorsque le programme *termine*, par définition cet arbre peut être *complété* pour donner un *arbre de trace complet*. L'application du Lemme 4.2.5 assure alors que m est un *mot de contrôle*. □

Concernant Queens: les *variables d'induction* sont n, i, j et k. La variable n reste en fait constante et la valeur de i n'a pas de rôle dans les *références-mémoire*. Seuls j et k seront donc considérées. On notera qu'un *mot de contrôle* terminé par s n'est pas concerné par la variable j.

4.3 Mots de contrôle et adresses-mémoire

4.3.1 Occurrences et cellules-mémoire

La fonction d'accès

Une *instruction-source* peut contenir zéro, une ou plusieurs *références*. Au niveau de l'exécution (une *trace* du programme), nous appellerons *accès* une *référence* dans une *occurrence*, c.-à-d. un couple *(occurrence, référence dans l'instruction)*. Un *accès* est soit *en lecture*, soit *en écriture*.

La *fonction d'accès* d'une *trace* associe à chacun de ses *accès* la valeur de la référence de l'*accès*, c.-à-d. une valeur d'adresse. Celle-ci est calculable à l'aide de la valeur des *variables d'induction*.

Une *valeur d'induction* est la valeur d'une *variable d'induction* en une *occurrence* du programme.

4.3.2 Mots de contrôle et cellules-mémoire

Mot de contrôle et état-mémoire

Lemme 4.3

L'exécution d'un bloc d'instructions *est sans effet sur l'adressage-mémoire existant hors de ce* bloc.

Preuve:
Soit P un programme et e une des exécutions de P.

Soit $\mathtt{v}_1$ et $\mathtt{v}_2$ 2 *variables d'induction*, initialisés par les exécutions de *blocs* respectives B_1 et B_2 incluses dans e, telles que la valeur de $\mathtt{v}_1$ est utilisée pour initialiser $\mathtt{v}_2$. Le non-chevauchement des *blocs* (**Règle des blocs**.1) impose que B_1 englobe B_2.

Par induction, on en déduit que toute *variable d'induction* dont la valeur dépend de $\mathtt{v}_1$ directement ou indirectement (c.à-d. par l'intermédiaire d'une chaîne transitive d'initialisations de *variables d'induction*), meurt avant B_1, ou plus tard lors de la fermeture de B_1. L'exécution de B_1 n'a donc aucune influence hors de B_1 sur les *variables d'induction* donc sur l'adressage-mémoire. □

Cela permet de revisiter *l'oubli*; la propriété qui suit est la pierre de touche de la modélisation.

Théorème 4.3

Soit u *un mot de contrôle, i* l'instance *associée et O_i l'ensemble des* occurrences *appartenant à i. Etant donné un élément o de O_i,* le tuple des valeurs d'induction *à l'issue de o ne dépend que de i. A chaque mot de contrôle ou indifféremment à chaque instance est donc affecté un tuple de* valeurs d'induction *unique.*

Preuve:
soit o une occurrence de l'exécution e d'un programme P. La valeur $\overrightarrow{V}$ de l'adressage-mémoire à l'issue de o est indépendant des *blocs d'instructions* exécutés. Transposé en termes de **labels**, cela veut dire que $\overrightarrow{V}$ ne dépend que du *mot de contrôle* relatif à o et non du *mot de trace*. Finalement, $\overrightarrow{V}$ est déterminée par i. □

On peut donc parler de **la valeur** d'une *variable d'induction* **à l'instance** i, ou encore au **mot de contrôle** u. Cela justifie l'introduction de la définition suivante:

Définition 4.5
Etant donnée une variable d'induction `v` *d'un programme* P,
une liaison *de* `v` *est un couple* (mot de contrôle, valeur de `v` au mot de contrôle),
la fonction de liaison *de* `v`, *notée* $\Lambda_{\mathtt{v}}$, *est l'ensemble des liaisons de* `v`,
la fonction de liaison $\overrightarrow{\Lambda}$ *de* P *est le tuple des* fonctions de liaison *pour toutes les* variables d'induction *de* P.

$$\overrightarrow{\Lambda} \triangleq (\Lambda_{\mathtt{v}_1}, \ldots, \Lambda_{\mathtt{v}_n})$$

Ainsi, le terme $\Lambda_{\boldsymbol{v}}(u)$ désigne la valeur de `v` pour le *mot de contrôle* u.

Relier accès et cellules-mémoire

On remarquera que la *fonction de liaison* $\overrightarrow{\Lambda}$ est indépendante d'une exécution particulière. Un *mot de contrôle* étant donné, lui est associée une *instance* bien définie. Si l'on considère une *occurrence* qui la représente dans une exécution particulière du programme, elle est identifiable sans ambiguïté par le *mot de contrôle*. Les *valeurs d'induction* associées sont fournies par la *fonction de liaison* du programme. La *fonction d'accès* du programme s'en déduit directement. On en conclut à bon droit que l'on a affaire à une analyse *par instances*.

Deuxième partie

Calcul des adresses-mémoire

Chapitre 1

Bi-labels

1.1 Motivation

Les 2 sections qui suivent ont pour projet la mise en place d'une technique de calcul de la *fonction de liaison*. En préliminaire, nous supposerons que nous disposons de l'automate fini des *mots de contrôle* du programme, ce qui nous permet d'accéder à une *expression rationnelle* du langage généré.

Rappelons que l'opération de *concaténation*, conventionnellement multiplicative, permettra de représenter l'évolution des *variables d'induction* au cours de l'exécution. Nous devrons construire des transductions qui produisent les valeurs des *variables d'induction* en toute *instance* du programme. La ressource de *l'automate de contrôle* nous invite à formuler ces résultats commme des expressions rationnelles, dont l'existence est assurée a posteriori par preuve constructive.

Nous proposons 2 méthodes équivalentes pour atteindre cet objectif.

La première, développée en Section 4, fait appel à la forme matricielle; la seconde, en Section 5, affecte un état distinct à chaque *variable d'induction* pour chaque point de programme. Les étiquettes des transductions seront nommées des *bi-labels*.

1.2 Les monoïdes de bi-labels

Rappelons que Σ_{ctrl} est l'alphabet des **labels** qui étiquètent les instructions.

Définition 1.1
Soit M_{adr} *le monoïde des valeurs de* n variables d'induction $(\mathrm{v}_k)_{\ 1\leqslant k\leqslant n}$.
Un bi-label *est un élément de l'ensemble* $\Sigma^*_{ctrl} \times M_{adr}$, *noté* B.

Interprétation sémantique

Un couple *(label d'instruction, translation associée de la valeur d'induction)*, de même qu'une *liaison (Défn I.4.5)*, sont des *bi-labels*.

Les *bi-labels* sont les éléments des transductions dont nous cherchons une expression rationnelle. Ils constituent la structure de données sur laquelle porteront nos calculs. Ils étiquètent les transitions des automates ou des transducteurs que nous allons construire, soit individuellement (en Section 5), soit en tant qu'éléments de matrice (en Section 4). Les opérations de modification des *liaisons* des *variables d'induction* à l'exécution travaillent sur les *bi-labels*.

Nous dotons B d'une structure de monoïde par l'opération de concaténation ($\bullet$) des *bi-labels* définie comme la concaténation parallèle des **labels** d'instruction et des *valeurs d'induction*. Si '$\cdot$' représente l'opération du monoïde M_{adr}, '$\bullet$' est donc définie comme suit:

$$(\alpha|a) \bullet (\beta|b) \triangleq (\alpha\beta|a \cdot b)$$

L'élément neutre de '$\bullet$' est $(\epsilon|\epsilon)$.

L'interprétation de '•' est la suivante:
Soit v une *variable d'induction* de valeur a à *l'instance* i dont le *mot de contrôle* associé est α. $(\beta|b)$ représente une instruction de **label** β qui incrémente a de la valeur b.

Remarque 1.1
Supposons que le programme gère des variables d'induction *qui prennent leurs valeurs dans plusieurs monoïdes, par exemple les entiers pour des adresses de tableaux à une dimension et le langage rationnel* $\{l, r\}^*$ *pour des adresses d'arbres binaires.*
Les calculs sur les variables d'induction *sont internes à chacun des monoïdes et nous pourrons donc les considérer* ***de manière totalement indépendante*** *pour chaque monoïde. B dénotera l'un quelconque de ces monoïdes.*

Remarque 1.2
Certaines instructions sont sans effet sur les variables d'induction. *La valeur d'induction incrémentale est alors l'élément neutre du monoïde.*

1.3 Les semi-anneaux d'expressions rationnelles de bi-labels

1.3.1 Construction

Nous munissons **l'ensemble des parties de** B de l'opération additive d'union et de l'opération multiplicative de concaténation induite de '•'.
Soit B_{il} l'ensemble des *parties rationnelles* de B : $B_{il} \triangleq Rat[\Sigma^*_{ctrl} \times M_{adr}]$
En vertu du Théorème B.1 en Annexe, B_{il} est un *semi-anneau.*
L'élément neutre de l'addition est $\emptyset$, l'ensemble vide de *bi-labels.* C'est aussi l'élément nul (absorbant) de la multiplication.

1.3.2 Bi-label représentatif d'une expression rationnelle

Expression rationnelle à l'intérieur d'un bi-label
Nous introduisons un abus de notation qui transgresse la remarque suivante:

Remarque préliminaire 1.3
L'opération additive d'union est une opération primitive, contrairement à '•' qui est induite de '·'.

Nous noterons cependant:

$$(\alpha + \beta|a) \triangleq (\alpha|a) + (\beta|a)$$

Dans ce cas, le *bi-label* en membre gauche de l'égalité est une expression rationnelle représentant un ensemble de *bi-labels* dont la valeur de sortie est identique et nous avons:

$$(\alpha|a), (\beta|a) \in B \Rightarrow (\alpha + \beta|a) \in B_{il}$$

Symétriquement:

$$(\alpha|a + b) \triangleq (\alpha|a) + (\alpha|b)\ , \qquad (\alpha|a), (\alpha|b) \in B \Rightarrow (\alpha|a + b) \in B_{il}$$

Bi-label représentant un singleton
En référence à la tradition des langages réguliers, nous confondrons un singleton avec le *bi-label* qu'il contient. Ainsi, $\{(s|\epsilon)\}$ sera noté $(s|\epsilon)$.

1.4 Vecteur de liaison

1.4.1 Définition

Notre objectif est le calcul de la *fonction de liaison* $\overrightarrow{\Lambda}$ du programme par ses parties indépendantes associées à chaque domaine de valeur M_{adr} du programme. Nous devons donc diviser $\overrightarrow{\Lambda}$ en autant de parties qu'il y a

de domaines de valeurs des *variables d'induction.* Ces sous-tuples de $\overrightarrow{\Lambda}$ seront appelés *vecteur de liaison.* Ils seront à évaluer à l'issue de chaque instruction du programme.

Définition 1.2

Soit $(\mathtt{v}_k)_{\,1\leqslant k\leqslant n}$ *le tuple des* variables d'induction *à valeurs dans le monoïde noté* M_{adr}*; nous l'appellerons* vecteur d'induction.

On appellera vecteur de liaison après l'instruction s*, noté* $\overrightarrow{L_s}$ *(respectivement: après* l'instance i*, noté* $\overrightarrow{L_i}$*), un tuple qui dénotera, pour chaque* $\mathtt{v}_k$*, l'ensemble des* liaisons *relatives chacune à une* instance *possible de* s *(respectivement: la* liaison *relative à* i*).*

Soit W_s l'ensemble des *mots de contrôle* terminés par le **label** de s:

$$\overrightarrow{L_s} \triangleq \left(\bigcup_{u\in W_s} \Lambda_{\boldsymbol{v}_1}(u),\ \ldots\ ,\ \bigcup_{u\in W_s} \Lambda_{\boldsymbol{v}_n}(u) \right),\quad \overrightarrow{L_s} \in {B_{il}}^n$$

$$\overrightarrow{L_i} \triangleq (\Lambda_{\boldsymbol{v}_1}(u),\ \ldots\ ,\ \Lambda_{\boldsymbol{v}_n}(u))\quad ,\quad \overrightarrow{L_i} \in B^n$$

Le terme '$\bigcup_{u\in W_s} \Lambda_{\boldsymbol{v}_k}(u)$' sera naturellement appelé *fonction de liaison de* $\mathtt{v}_k$ *après l'instruction* s et noté $\Lambda^{s}_{\boldsymbol{v}_k}$ ou plus simplement $\Lambda_{\boldsymbol{v}_k}$ si aucune confusion n'en résulte.

1.4.2 Interprétation

Nous calculerons $\overrightarrow{L_s}$ comme un élément de ${B_{il}}^n$, c.-à-d. comme un vecteur composé d'expressions rationnelles de *bi-labels.* Il est important de noter que *l'automate de contrôle* est l'outil adapté à ce projet. En effet, d'après le Théorème II.4.3, l'évolution des *valeurs d'induction* est parallèle à celle des *mots de contrôle.*

Le calcul de $\overrightarrow{L_i}$ après *l'instance* i suit le chemin associé au *mot de contrôle*sur *l'automate de contrôle*, depuis le nœud $START$ jusqu'à i.

Plus globalement, le calcul du *vecteur de liaison* $\overrightarrow{L_s}$, c.-à-d. de l'ensemble des *mots de contrôle* possibles après l'instruction s avec les *valeurs d'induction* associées, se calque sur l'ensemble des chemins qui aboutit à s.

1.4.3 Addition des vecteurs de liaison

Nous munissons ${B_{il}}^n$ d'une opération additive d'union induite naturellement de B_{il}. Cette opération prolonge aux *vecteurs de liaison* l'opération d'union dans $Rat(\Sigma^*_{ctrl})$, l'ensemble des expressions rationnelles de mots.

Chapitre 2

Programme et automate matriciel

2.1 Matrices $n \times n$ d'expressions rationnelles de bi-labels

2.1.1 M_{bil} est un semi-anneau

Nous notons M_{bil} l'ensemble ${B_{il}}^{n \times n}$ des matrices carrées de dimension n d'éléments de B_{il}. Dans cette section, nous désirons modéliser les calculs portant sur les expressions rationnelles de *bi-labels* à l'aide de la forme matricielle.

Le théorème suivant nous y invite, car il nous ouvre la voie du calcul dans M_{bil}.

Théorème 2.1
l'ensemble M_{bil} des matrices de dimension n à éléments dans B_{il} est un semi-anneau.

Preuve:
La preuve découle immédiatement de l'application du Théorème A.1. □

2.1.2 Représentation des opérations dans M_{bil}

Nous montrons maintenant comment représenter les opérations autorisées sur les variables d'induction.

Les opérations

Règle 2.1
Afin de conserver l'ordre séquentiel de production et de lecture des mots de gauche à droite, nous adoptons également ce mode de représentation pour les matrices.

La Règle 2.1 impose de représenter le *vecteur de liaison* avant l'instruction **en ligne et à gauche** de la matrice.

Opération et matrice
Une *opération* est une modification des *variables d'induction* par une instruction ou une séquence d'instructions. A toute opération s de notre modèle de programme est associée une matrice, notée Δ_s.

Celle-ci transforme un *vecteur de liaison* donné en entrée conformément à l'opération s.

Le type d'une opération est: $s : {B_{il}}^n \rightarrow {B_{il}}^n$

Dans le mode **en ligne**, la matrice Δ_s est la transposée de la matrice d'application linéaire habituelle associée à s.

Déroulement des calculs
Soit $\overrightarrow{L}_{s_0}$ le *vecteur de liaison* exprimant l'ensemble des *liaisons* possibles après l'instruction s_0. Soit s_1 un des successeurs de s_0 sur *l'automate de contrôle.*

L'ensemble des *liaisons* possibles à l'issue de la séquence $s_0 s_1$ est: $\overrightarrow{L}_{s_0 s_1} = \overrightarrow{L}_{s_0} \times \Delta_{s_1}$

Le *vecteur de liaison* en s_1 est l'union des contributions individuelles de tous les prédécesseurs s_0 de s_1 sur *l'automate de contrôle.*

On remarquera que la morphologie des calculs est isomorphe à celle du langage rationnel des *mots de contrôle* reconnu par *l'automate de contrôle.*

Exemple

Pour illustrer notre propos, nous supposons le *vecteur de liaison* construit sur 3 variables i, j et k. Rappelons que chaque composante du *vecteur de liaison* est une expression rationnelle de *bi-labels*.

- *remise à zéro*

  On ajoute au *vecteur d'induction* une coordonnée spéciale z de valeur constante égale à ϵ, l'élément neutre de M_{adr}.

  Soit $\overrightarrow{L_s} = (\Lambda_i^s, \Lambda_j^s, \Lambda_k^s, \Lambda_z^s)$ le *vecteur de liaison* après l'instruction s. La matrice et l'opération matricielle associées à la *remise à zéro* de j par une instruction s_1 qui suit s sont:

$$(\Lambda_i^s, \Lambda_j^s, \Lambda_k^s, \Lambda_z^s) \begin{bmatrix} s_1|\epsilon & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & s_1|\epsilon & \emptyset \\ \emptyset & s_1|\epsilon & \emptyset & s_1|\epsilon \end{bmatrix} = (\Lambda_i^s \bullet (s_1|\epsilon),\ \Lambda_z^s \bullet (s_1|\epsilon),\ \Lambda_k^s \bullet (s_1|\epsilon),\ \Lambda_z^s \bullet (s_1|\epsilon))$$

  Rappelons que nous travaillons dans B_{il}. Les *bi-labels* des *vecteurs de liaison* dénotent des expressions rationnelles et les *bi-labels* de la matrice sont des singletons.

- *translation*

$$(\Lambda_i^s, \Lambda_j^s, \Lambda_k^s, \Lambda_z^s) \begin{bmatrix} s_1|a & \emptyset & \emptyset & \emptyset \\ \emptyset & s_1|\epsilon & \emptyset & \emptyset \\ \emptyset & \emptyset & s_1|c & \emptyset \\ \emptyset & \emptyset & \emptyset & s_1|\epsilon \end{bmatrix} = (\Lambda_i^s \bullet (s_1|a),\ \Lambda_z^s \bullet (s_1|\epsilon),\ \Lambda_k^s \bullet (s_1|c),\ \Lambda_z^s \bullet (s_1|\epsilon))$$

- *capture*

$$(\Lambda_i^s, \Lambda_j^s, \Lambda_k^s, \Lambda_z^s) \begin{bmatrix} \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & s_1|\epsilon & \emptyset & \emptyset \\ s_1|\epsilon & \emptyset & s_1|\epsilon & \emptyset \\ \emptyset & \emptyset & \emptyset & s_1|\epsilon \end{bmatrix} = (\Lambda_k^s \bullet (s_1|\epsilon),\ \Lambda_j^s \bullet (s_1|\epsilon),\ \Lambda_k^s \bullet (s_1|\epsilon),\ \Lambda_z^s \bullet (s_1|\epsilon))$$

- *initialisation*

$$(\Lambda_i^s, \Lambda_j^s, \Lambda_k^s, \Lambda_z^s) \begin{bmatrix} s_1|\epsilon & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & s_1|\epsilon & \emptyset \\ \emptyset & s_1|b & \emptyset & s_1|\epsilon \end{bmatrix} = (\Lambda_i^s \bullet (s_1|\epsilon),\ \Lambda_z^s \bullet (s_1|b),\ \Lambda_k^s \bullet (s_1|\epsilon),\ \Lambda_z^s \bullet (s_1|\epsilon))$$

- *écrasement*

$$(\Lambda_i^s, \Lambda_j^s, \Lambda_k^s, \Lambda_z^s) \begin{bmatrix} s_1|\epsilon & \emptyset & \emptyset & \emptyset \\ \emptyset & s_1|\epsilon & s_1|b & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & s_1|\epsilon \end{bmatrix} = (\Lambda_i^s \bullet (s_1|\epsilon),\ \Lambda_j^s \bullet (s_1|\epsilon),\ \Lambda_j^s \bullet (s_1|b),\ \Lambda_z^s \bullet (s_1|\epsilon))$$

- *mixture*

$$(\Lambda_i^s, \Lambda_j^s, \Lambda_k^s, \Lambda_z^s) \begin{bmatrix} \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & c & \emptyset \\ s_1|a & \emptyset & \emptyset & \emptyset \\ \emptyset & s_1|b & \emptyset & s_1|\epsilon \end{bmatrix} = (\Lambda_k^s \bullet (s_1|a),\ \Lambda_z^s \bullet (s_1|b),\ \Lambda_j^s \bullet (s_1|c),\ \Lambda_z^s \bullet (s_1|\epsilon))$$

Interprétation

Le *vecteur de liaison* après l'instruction s_1 est l'union des contributions de chacune des instructions qui précèdent possiblement s_1, c.à-d. qui précèdent s_1 sur *l'automate de contrôle*.

Pour **Queens** par exemple, l'instruction de **label** G est précédée des intructions de **labels** respectifs F et f.

Un élément de matrice δ_{ij} porte une contribution de la *variable d'induction* i à l'expression rationnelle des nouvelles *liaisons* de la *variable d'induction* j. $\emptyset$ traduit une contribution nulle, ϵ signifie que les valeurs de i sont transmises sans changement à j. Une valeur a révèle que les valeurs de i incrémentées de a sont transmises à j.

2.2 Automate matriciel

Définition 2.1

Un automate matriciel *est un automate dont les étiquettes sont des* bi-labels *particuliers: leur alphabet de sortie est constitué de matrices.*

2.2.1 Elaboration de l'automate

Notre but est d'exprimer l'évolution parallèle des *mots de contrôle* et des *variables d'induction* d'un programme ou d'une partie de programme sous la forme de *transductions séquentielles*. Celles-ci seront les composantes d'une matrice de M_{bil}. Cette dernière sera élaborée à l'aide d'un automate construit comme suit:

Sur *l'automate de contrôle*, on substitue au **label** de chaque transition la matrice des *bi-labels* représentative de l'opération effectuée sur le *vecteur de liaison* par l'instruction.

On peut donc obtenir *à la volée* (séquentiellement) les mots matriciels reconnus par l'automate. Ces mots expriment la transformation réalisée sur le *vecteur de liaison* depuis le début du programme.

L'ensemble des mots matriciels reconnus pour un programme P forme ainsi un langage rationnel, noté $L_P \in Rat(M_{bil})$, sur un alphabet inclus dans M_{bil}.

Remarques

- *L'automate matriciel* AM contient toute l'information pour déterminer la fonction de liaison $\overrightarrow{\Lambda}$. En effet, le domaine de $\overrightarrow{\Lambda}$ est l'ensemble L_{ctrl} des *mots de contrôle* et les matrices qui décorent AM expriment l'évolution de $\overrightarrow{\Lambda}$ pour chaque *variable d'induction* et pour chaque *instance* du programme.
- Une démarche équivalente à l'utilisation de *l'automate* consiste en la substitution de la matrice correspondant à chacune des lettres de l'expression rationnelle décrivant les *instances* en un point de programme.
- Une expression rationnelle dans $Rat(M_{bil})$ ne nous contente pas, l'objectif que nous poursuivons est de trouver une matrice de M_{bil}.

2.2.2 L'automate matriciel de Queens

Le *vecteur d'induction* du programme est (j, k, z). Nous le supposons initialisé à $(\epsilon, \epsilon, \epsilon)$. Le *vecteur de liaison* est donc $(\{\epsilon|j\}, \{\epsilon|k\}, \{\epsilon|z\})$. Ici, ϵ est représentée par la valeur entière 0. Nous notons $\left[\epsilon\right]$ la matrice *Identité*. Par abus de notation mais pour améliorer la lisibilité de la figure, le **label** d'entrée est écrit hors des *bi-labels*.

2.3 La fonction de liaison

2.3.1 Calcul de *l'étoile* d'une matrice

Une expression rationnelle X dans M_{bil} ne nous satisfait pas. Or, l'opérateur *étoile* sur les matrices est a priori un obstacle pour transformer X en une matrice de M_{bil}. Rappelons que cette dernière nous livrerait la valeur du *vecteur de liaison*, qui fournit **une expression rationnelle des *liaisons* pour chaque variable d'induction**, comme nous l'avons expliqué en Section 2.1.2.

C'est pourquoi nous montrons maintenant comment calculer *l'étoile* d'une matrice A. Nous présentons 2 techniques équivalentes.

Résolution par équations récurrentes

A^* est une solution de l'équation:

$$Y = A \times Y + \epsilon, \quad | \quad \epsilon\text{: matrice Identité} \tag{2.3.1}$$

C'est un système de n^2 équations linéaires droites à n^2 expressions rationnelles inconnues, **résoluble par susbstitutions successives**. Il en existe une plus petite solution (au sens de l'inclusion), c'est-à-dire telle que chaque expression soit la plus petite solution. Cette solution est A^*.

- A^* est solution : *trivial*

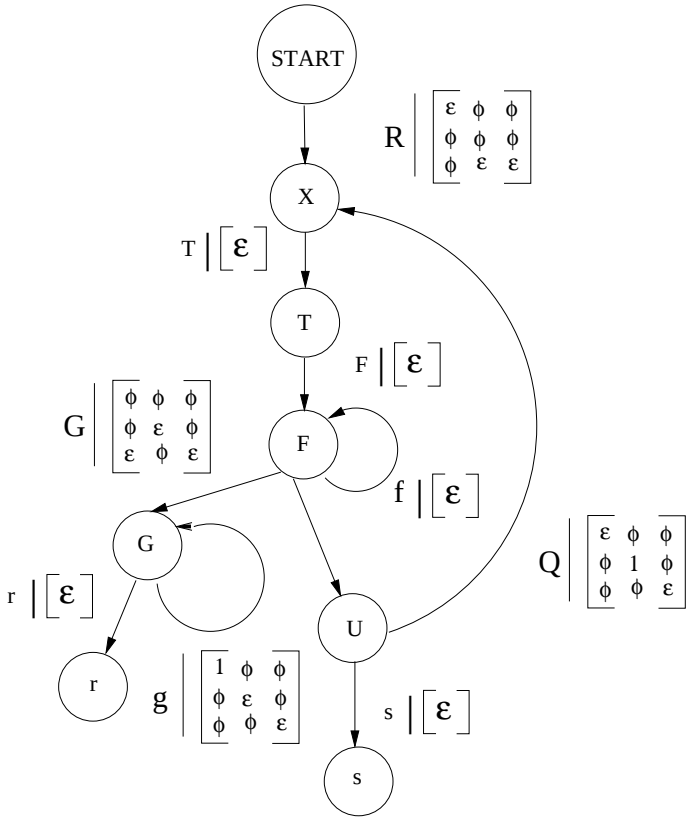

Figure 6. L'automate matriciel du programme **Queens**

– c'est la plus petite:

  – ϵ est inclus dans toute solution

  – on en déduit par substitution en chaîne que A est inclus dans toute solution, puis A^2, A^3, donc A^*.

□

Résolution par tracé d'un automate

Nous allons utiliser la représentation matricielle d'un transducteur que nous détaillons plus loin ($Sect^n$ 4.2).

A titre d'exemple, cherchons le *produit étoilé* d'une matrice à éléments rationnels en dimension 2.

Nous dessinons un automate dont les états sont 2 coordonnées, i et j. Les transitions représentent la transmission des valeurs entre ces 2 variables lors de l'application de la matrice.

La transformation opérée:

$$(i_0, j_0) \begin{bmatrix} a & c \\ b & d \end{bmatrix} = (i_0 a + j_0 b,\ i_0 c + j_0 d) \tag{2.3.2}$$

est simulée par l'automate de la Figure 7.

Il est aisé de s'apercevoir que la fermeture transitive de cette opération conduit à la matrice présentée sur la Figure.

2.3.2 Calcul de la fonction de liaison

Le programme prend ainsi la forme d'une suite d'opérations matricielles, c'est-à-dire d'une expression algébrique E_m dans M_{bil} combinant des opérations d'addition (alternative), de produit (séquence) et de l'étoile (boucle ou appel récursif) dans M_{bil}.

Rappelons que nous disposons de l'expression rationnelle, dans le langage L_{ctrl} des *mots de contrôle*, représentative du programme ou d'une *instance*.

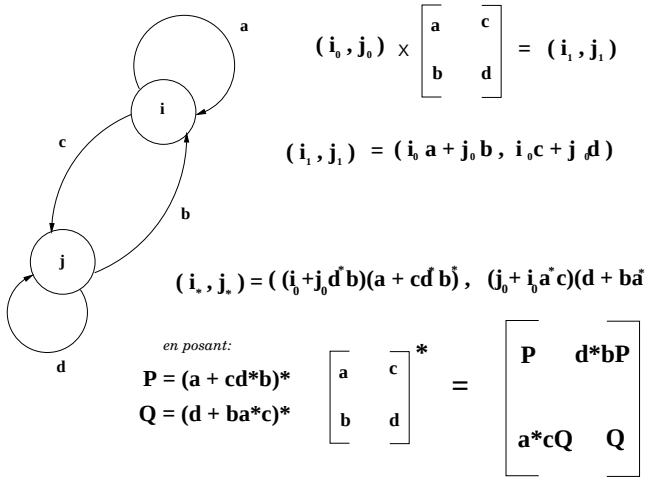

Figure 7 . L'automate simulant *l'étoile d'une matrice*

L'expression E_m est le résultat d'une substitution appliquée à l'expression rationnelle E du programme: à chaque lettre l de E est substituée la matrice de M_{bil} associée à l'opération dont l est le **label**, comme nous l'avons explicité en Section 2.1.2.

Soit $[L]$ la matrice-résultat de ce calcul jusqu'à un point de programme P. Un élément L_{ij} de cette matrice représente la contribution des chemins dont le point de départ, au début du programme, est l'état attaché à la *variable d'induction* de rang i et qui aboutissent au point de programme P sur l'état attaché à la *variable d'induction* de rang j.

Si $\overrightarrow{V_0}$ est la *liaison (Défn I.4.5)* initiale, le produit $\overrightarrow{V_0} \times [L]$ fournit la *fonction de liaison.*

2.3.3 La fonction de liaison de Queens

Expression matricielle

Rappelons que le *vecteur d'induction* du programme est (j, k, z), où z est la variable spéciale d'initialisation. Comme indiqué en Section 2.2.1, nous substituons dans l'expression rationnelle les *bi-labels* aux **labels**.

Les matrices apparaissent en figure 3. En partant de l'expression 4.2.1:

$$w = \mathrm{RTF(f + UQTF)^*(Us + Gg^*r)}$$

on obtient l'expression rationnelle matricielle:

$$\begin{bmatrix} RTF|\epsilon & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset \\ \emptyset & RTF|\epsilon & RTF|\epsilon \end{bmatrix} \times \left(\begin{bmatrix} f|\epsilon & \emptyset & \emptyset \\ \emptyset & f|\epsilon & \emptyset \\ \emptyset & \emptyset & f|\epsilon \end{bmatrix} + \begin{bmatrix} UQTF|\epsilon & \emptyset & \emptyset \\ \emptyset & UQTF|1 & \emptyset \\ \emptyset & \emptyset & UQTF|\epsilon \end{bmatrix} \right)^* \times \cdots$$

$$\left(\begin{bmatrix} Us|\epsilon & \emptyset & \emptyset \\ \emptyset & Us|\epsilon & \emptyset \\ \emptyset & \emptyset & Us|\epsilon \end{bmatrix} + \begin{bmatrix} \emptyset & \emptyset & \emptyset \\ \emptyset & G|\epsilon & \emptyset \\ G|\epsilon & \emptyset & G|\epsilon \end{bmatrix} \times \begin{bmatrix} g|\epsilon & \emptyset & \emptyset \\ \emptyset & g|\epsilon & \emptyset \\ \emptyset & \emptyset & g|\epsilon \end{bmatrix}^* \times \begin{bmatrix} r|\epsilon & \emptyset & \emptyset \\ \emptyset & r|\epsilon & \emptyset \\ \emptyset & \emptyset & r|\epsilon \end{bmatrix} \right).$$

Calcul et résultat final

On remarque que les matrices *étoilées* sont diagonales pour leurs *mots de sorties* et donc immédiatement calculables. La matrice finale $[L]$ se calcule sans difficulté. On pose les notations 2.3.3:

$$\alpha = RTF \ , \ \beta = UQTF \tag{2.3.3}$$

$$[L] = \begin{bmatrix} \alpha(f+\beta)^*Us|\epsilon & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset \\ (\alpha(f+\beta)^*G|\epsilon)(g|1)^*(r|\epsilon) & (\alpha|\epsilon)((f|\epsilon)+(\beta|1))^*(Us+Gg^*r|\epsilon) & \alpha(f+\beta)^*(Us+Gg^*r)|\epsilon \end{bmatrix}$$

Interprétation

Comme indiqué en Section 2.1.2, la *fonction de liaison* est fournie par la formule:

$$\overrightarrow{\Lambda} = (\epsilon|j_0, \epsilon|k_0, \epsilon|\epsilon) \times [L] \quad , \qquad (j_0, k_0) : \textit{vecteur d'induction initial} \tag{2.3.4}$$

La *fonction de liaison* Λ_j est donc donnée par $(\epsilon|j_0) \bullet L_{11} + L_{31}$, et Λ_k par L_{32}.

En pratique la portée de j ne touche pas les *mots de contrôle* exprimés dans L_{11}. Seule L_{13} dénote la *fonction de liaison* Λ_j. Les *variables d'induction* étant ré-initialisées par les opérations du programme, il est logique que leurs valeurs dans le *vecteur d'induction initial* n'apparaissent plus au résultat.

Bilan de la méthode matricielle

La technique que nous venons d'illustrer avec le programme **Queens** nécessite des calculs avec des matrices creuses. De plus, on peut remarquer le supplément de calcul introduit avec la variable z, ainsi que le rejet d'une partie des résultats comme non signifiants. La technique développée dans la Section suivante, outre qu'elle s'avère plus économe, est plus intuitive car elle visualise le calcul de chaque *variable d'induction* séparément.

Chapitre 3

Programme et transducteur vectoriel

3.1 Transducteur vectoriel (TV)

3.1.1 Définition

Définition 3.1

Soit E un ensemble, dont les éléments seront appelés des états. *Un* état vectoriel *de dimension n est un vecteur (ou n-uplet) composé de n* états.

*Soit V un ensemble d'*états vectoriels *de dimension n.*

Un transducteur vectoriel *est un* transducteur *qui satisfait aux contraintes suivantes:*

- *les états sont les composantes des vecteurs de V, auxquels on ajoute un état initial START afin de respecter la définition du transducteur (Déf^n C.4);*
- *comme les états, les transitions forment des n-uplets appelés* transition vectorielle*; les composantes d'une même* transition vectorielle *ont pour origine (respectivement: pour destination) le même* état vectoriel*;*
- *les transitions d'une même* transition vectorielle *portent la même étiquette d'entrée.*

3.1.2 Interprétation

Chaque coordonnée incarne une *variable d'induction.* Dans le cas général, chaque *état vectoriel* $\overrightarrow{V_s}$ incarne une instruction s. Le **label** de s étiquètera les composantes de $\overrightarrow{V_s}$ sur le graphe du *transducteur.* Il y a 2 exceptions: pour la boucle, pour laquelle un unique *état vectoriel* regroupera l'instruction d'entrée dans la boucle (***bloc_for***) et l'instruction d'entrée dans les itérations ultérieures (**itération_plus**↓), et pour la procédure, pour laquelle un *état vectoriel* regroupe la déclaration et les appels à cette procédure. Un état de coordonnée v dans $\overrightarrow{V_s}$ dénote le calcul de la *liaison* de v pour l'ensemble des *instances* terminées sur s.

Eu égard au dernier critère de la Définition 3.1, le TV est un *transducteur non séquentiel* (Déf^n C.6).

3.2 Evolution du vecteur d'induction

3.2.1 Les transitions

L'appareillage du *transducteur vectoriel* est dédié à la représentation de l'exécution du programme *modulo les instances.*

L'observation de la Figure 8, réalisée pour **Queens**, donne l'intuition de la construction.

Notre *transducteur vectoriel* est bâti sur les fondations de *l'automate de contrôle.* Le *label* d'une transition est un *bi-label* :
(**label** de l'instruction, contribution de l'état-départ à l'état-arrivée).

Une transition, de **label** (s, α), depuis une composante $\mathtt{u}_i$ de *l'état vectoriel-départ* $\overrightarrow{U}$ vers une composante $\mathtt{v}_j$ de *l'état vectoriel-arrivée* $\overrightarrow{V}$, exprime que la valeur v_j de $\mathtt{v}_j$ à l'état $\overrightarrow{V}$ est issue de u_i suivant la formule:

$$v_j = u_i \bullet \alpha \tag{3.2.1}$$

L'état $START$ est lié à $\overrightarrow{V_0}$, l'état vectoriel existant à l'initialisation du programme par une transition vers chaque composante V_i, de label (ϵ, V_{0_i}). Nous appelons $INIT$ l'état-arrivée recevant cette transition vectorielle. Cette initialisation peut être considérée comme une instruction du programme. Nous ne l'avons pas explicitée dans les Sections précédentes.

Les états *d'acceptation* sont choisis selon les objectifs du calcul. Dans le cas de **Queens**, la phase d'initialisation est sans effet.

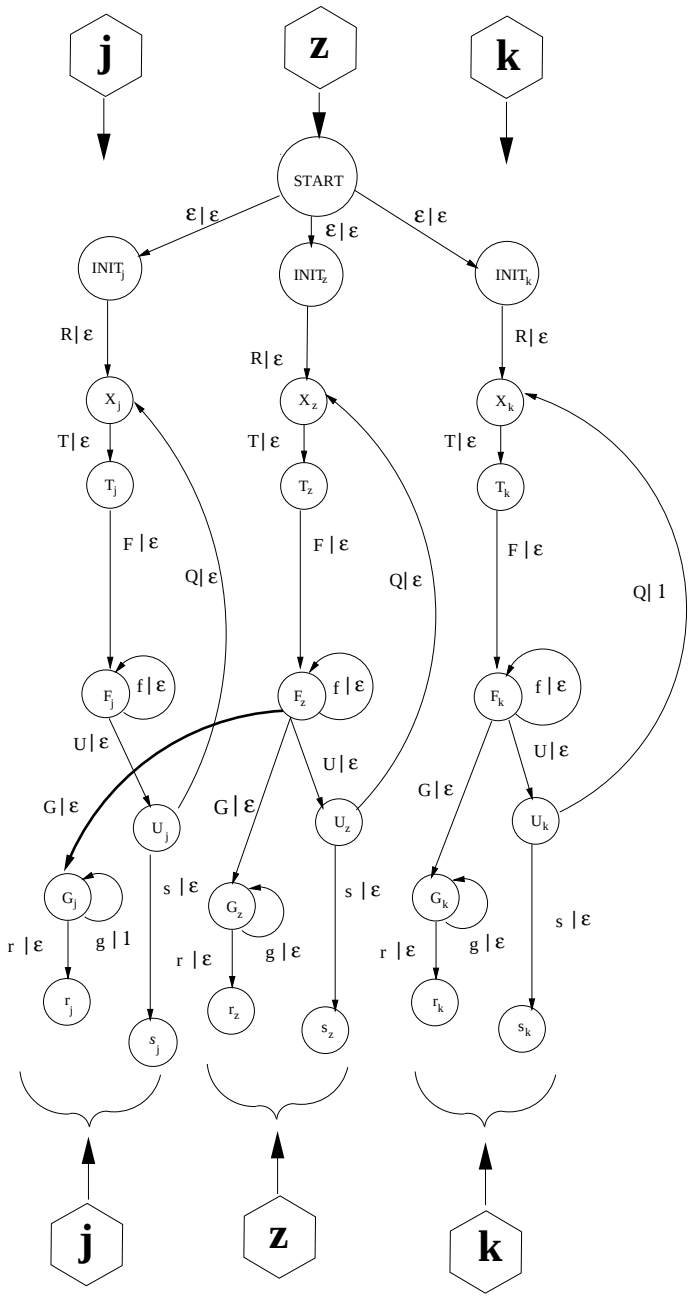

Figure 8. Le transducteur vectoriel du programme **Queens**

3.2.2 Les opérations

Ce modèle réalise les opérations autorisés (Section 2.5). Il suffit de le vérifier pour les *opérations de base*. La *remise à zéro* sera examinée dans la Section suivante.

- Notre modèle autorise la coordonnée i à distribuer sans modification sa valeur à la coordonnée j lors d'une transition t, selon la formule 3.2.1, avec $\alpha = \epsilon$. C'est l'opération de *capture*.
- Lorsque le *mot de sortie* n'est pas le mot vide, l'opération est une *translation*, une *initialisation* ou un *écrasement*.

3.2.3 Remise à zéro d'une variable d'induction

On souhaite ré-initialiser une coordonnée i lors de l'instruction R_z.

On ajoute au *vecteur d'induction* une coordonnée spéciale notée z et donc un nouvel état noté Z à chaque état vectoriel. Toutes les instructions du programme se contentent de transmettre *l'élément neutre* ϵ (soit le mot vide) sur cette coordonnée, à l'aide d'une transition reliant les états Z de 2 instructions successives. L'instruction R_z implante en sus une transition depuis l'état Z de chaque instruction qui précède R_z vers l'état représentant la coordonnée i dans *l'état vectoriel* de R_z.
La ré-initialisation de la coordonnée i est donc simulée par un chemin d'arcs qui saute d'une occurence de Z à la suivante depuis le début de l'exécution du programme jusqu'avant la transition R_z. Artifices calculatoires, les états Z ne seront jamais *d'acceptation*.

3.2.4 Le transducteur vectoriel de Queens

L'application du modèle explicité dans cette Section fournit le *transducteur vectoriel* de la Figure 8.

3.3 Calcul de la fonction de liaison

3.3.1 Le principe

Le transducteur que nous venons de construire réalise la *fonction de liaison*. C'est le point de vue *statique* (voir Annexe C). La restriction de celle-ci sur un point de programme (respectivement: en sortie d'une instruction)) consiste à retenir uniquement *l'état vectoriel* (respectivement: la *transition vectorielle* associée) comme ensemble d'états finaux (respectivement: de transitions finales) du transducteur. De même, la projection de cette fonction sur une *variable d'induction* particulière peut être réalisée en ne retenant que les états correspondants à cette variable. Ces 2 paramétrages peuvent être combinés.

3.3.2 Application au programme Queens

On vérifie sans difficulté que les expressions obtenues par la méthode matricielle coïncident avec le langage reconnu par le présent transducteur.

3.3.3 Transducteur vectoriel émondé (TVE)

Définition 3.2
Un TVE est un transducteur vectoriel *qui possède un seul état d'acceptation et qui est* émondé *selon la Déf^n C.7.*

L'intérêt de l'émondage est dans le caractère *séquentiel* du transducteur résultant. Il calcule *à la volée* l'expression rationnelle donnant la valeur d'une *variable d'induction* choisie, en un point de programme fixé. La sélection des *variables d'induction* et des points de programme étaient pris en compte dans le TV par le choix de ses états *d'acceptation*. Avec le TV, le calcul *à la volée* avait de grandes chances de se terminer sur un état-*poubelle* (Déf^n C.8).

Le transducteur étant séquentiel, on remarquera que l'indiçage de ses états devient superflu.

Queens, pour l'exemple !
La Figure 9 montre les TVE instanciés respectivement pour les variables j au point de programme de **label** r, et k au point de programme de **label** s.

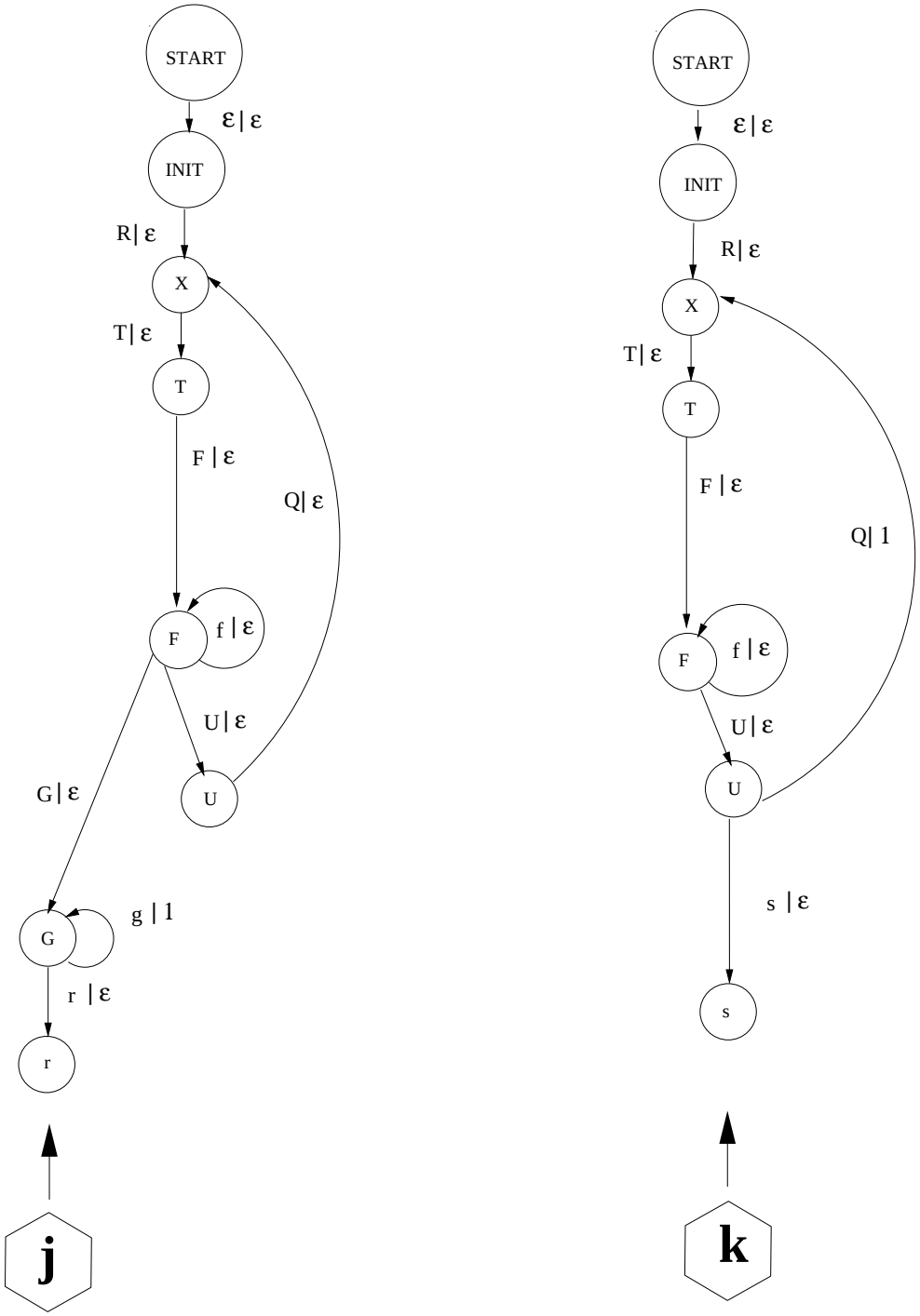

Figure 9. 2 transducteurs vectoriels émondés pour **Queens**

Chapitre 4

Mise en regard: *matriciel* versus *vectoriel*

Nous désirons montrer ici l'équivalence entre les démarches des 2 Sections précédentes. Ce rapprochement fait appel à la modélisation classique des *transductions rationnelles* par une représentation matricielle, introduite par Nivat [Niv68, Ber79, AB88]. Pour le montrer, nous devons mentionner auparavant une 3^e méthode de représentation de notre sujet, intermédiaire entre celles ci-dessus.

4.1 Transducteur compacté

Reprenons le *transducteur vectoriel* de la *Section 5*. Plutôt que de différencier les états suivant les instructions, nous fusionnons tous les *états vectoriels* pour n'en former plus qu'un. Il ne subsiste donc qu'un état pour chaque *variable d'induction*, plus l'état Z spécial de ré-initialisation. La *fonction de liaison* n'est plus le langage généré par le transducteur. Elle est obtenue en ne considérant qu'une partie de celui-ci, produite en sélectionnant comme *mot d'entrée* le *mot de contrôle* du programme ou de *l'instance* désirée. Nous passons de la vision *statique* à la vision *dynamique* (voir Annexe C).

Sur ce transducteur, nous allons construire la représentation matricielle classique.

4.2 Représentation matricielle classique

On modélise les productions d'un transducteur (*alphabétique* ou non) sous forme de matrices carrées à valeurs dans les *mots de sortie*. La dimension est le cardinal de **l'ensemble des états**.

A chaque *mot d'entrée* l du transducteur, on associe une matrice carrée $[L]$. L'élément L_{ij} est l'expression rationnelle dénotant l'ensemble des mots de sortie v tels que (l, v) soit l'étiquette d'un chemin d'origine l'état i et d'arrivée l'état j. La concaténation des chemins le long des arcs du transducteur est simulée par le produit matriciel tandis que l'addition fournit l'union des ensembles de *mots de sortie* pour l'ensemble des chemins *d'état-origine* i et *d'état-arrivée* j.

C'est cette représentation que nous avions utilisée à la Section 2.3.1 pour calculer *l'étoile* d'une matrice. Dans ce cas, c'est le caractère *séquentiel* de la transduction qui nous avait permis d'exécuter ce calcul *à la volée*.

4.3 Comparaison avec le transducteur matriciel

Notre transducteur de travail T est le *transducteur compacté* précédent modifié comme suit: ses *bi-labels* sont des couples (*label de l'instruction, bi-label associé à l'instruction*).

En appliquant le point de vue *dynamique* à T, (*Défn C*.4), le produit matriciel obtenu en sortie en suivant un *mot de contrôle* en entrée est identique à celui obtenu par *l'automate matriciel* pour le même *mot de contrôle*. En effet, les matrices construites par les 2 techniques sont identiques.

4.4 Le pont *matriciel* $\leftrightarrow$ *vectoriel*

Dans cette Section, nous désirons produire directement l'expression matricielle désirée depuis le *transducteur vectoriel* sans passer par le *transducteur compacté*: ainsi la connaissance préalable d'un *mot de contrôle* à suivre devient inutile. C'est le point de vue *statique* (*Défn C*.4).

Pour cela, nous présentons une représentation matricielle du transducteur plus compacte que la représentation classique: nous allons personnaliser la matrice associée à chaque *lettre d'entrée*. L'idée est développée dans la Section suivante.

4.4.1 Représentation matricielle compactée

Pour calculer les transductions, seuls comptent les chemins qui sont des *trajets* (*Déf*^{n} *C*.7). Nous pouvons donc restreindre l'étude à des transducteurs *émondés* (*ibidem*). Ainsi, chaque *état-arrivée* devient *état-départ* de la *transition vectorielle* suivante.

Dans la nouvelle présentation, une ligne d'une matrice Δ n'est pas attribuée à chaque *état* du transducteur, mais uniquement à **chaque *état-départ* de la *transition vectorielle*** représentée par Δ; de même, une ligne n'est attribuée qu'à **chaque *état-arrivée***. Ainsi, dans le cas général, cette représentation produit des matrices non obligatoirement carrées, mais qui acceptent cependant l'opération de produit.

4.4.2 Représentation matricielle du transducteur vectoriel

Cette formalisation appliquée au *transducteur vectoriel* permet de retrouver directement la modélisation de *l'automate matriciel*. Il suffit d'opérer une substitution identique à celle de la Section 4.3, c'est-à-dire de remplacer sur le *transducteur* chaque **label** de sortie s par le *bi-label* (e, s) où e est le *label d'entrée* correspondant.

Annexes

Annexe A

Monoïdes et semi-anneaux

Définition A.1
Un monoïde *est un ensemble muni d'une opération binaire associative avec un élément neutre.*

Définition A.2
Soit X un alphabet, nous rappelons la notation:

$$X^* \triangleq \bigcup_{n \geqslant 0} X^n, \qquad \textit{où l'opération } \text{`puissance n'} \textit{ est induite de la concaténation.}$$

X^* est appelé le monoïde libre engendré par X, ou de base X, *pour l'opération de concaténation.*

Définition A.3
Un semi-anneau *est un monoïde pour 2 opérations binaires, $+$, qui est commutative et $\times$, distributive par rapport à la première. De plus, l'élement neutre pour $+$ est* élément nul *ou* absorbant *pour $\times$.*

Théorème A.1
Soit S un semi-anneau. *L'ensemble $S^{n \times n}$ des matrices carrées d'éléments de S de dimension n est un* semi-anneau *pour les opérations d'addition et de multiplication matricielles induites.*

Annexe B

Partie, relation, expression rationnelles

Définition B.1
Soit M un monoïde. L'ensemble $Rat(M)$ des parties rationnelles *de M est le plus petit ensemble qui contient les parties finies, et fermé par union, produit et étoile.*

Définition B.2
Soit X et Y 2 aphabets.
Une relation rationnelle *sur X et Y est une* partie rationnelle *du monoïde $X^* \times Y^*$.*
Une relation reconnaisable *sur X et Y est une partie du monoïde $X^* \times Y^*$* reconnaissable *par un automate fini.*

Rappels sur les expressions rationnelles

Une expression rationnelle dénote un ensemble de mots sur un alphabet. Par exemple $a+b$ représente $\{a, b\}$, a^*b est l'ensemble des mots capables de s'unifier avec cette expression par instanciation de l'étoile.

Une lettre est confondue avec un mot de 1 lettre; un mot est confondu avec le singleton qui le contient.

L'ensemble des expressions rationnelles fondées sur un alphabet A forme un *semi-anneau* S_{rat}, muni de la loi commutative de l'addition $(+)$, qui représente l'union de 2 ensembles de mots, et de la concaténation $(\cdot)$, jouant le rôle de loi multiplicative. Nous noterons respectivement ϵ et $\emptyset$ l'élément neutre et l'élément nul pour '$\cdot$'. Le symbole '$\cdot$' pourra être omis.

Définition B.3
L'expression rationnelle *qui dénote une* partie rationnelle *d'un* monoïde *M est formée des lettres d'un alphabet A représentant les singletons du monoïde, combinées par les opérateurs d'union, produit et* étoile.

Théorème B.1
L'ensemble des parties d'un monoïde est un semi-anneau *pour l'union et le produit induit de l'opération du monoïde. L'ensemble de ses parties rationnelles en est un* sous-semi-anneau *isomorphe à l'ensemble des expressions rationnelles.*

Annexe C

Automates, transducteurs finis, grammaire algébrique

Définition C.1
Soit X et Y 2 alphabets. Une transduction, *notée $\tau : X^* \to Y^*$ est une fonction de X^* dans $\mathcal{P}(Y^*)$.*

Définition C.2
Le graphe *d'une transduction $\tau : X^* \to Y^*$ est la relation induite sur $X^* \times Y^*$.*
Une transduction est rationnelle *si son* graphe *est une* relation rationnelle.
Une transduction est reconnaissable *si son* graphe *est* une relation reconnaissable.

Le graphe d'une transduction en donne une photographie (point de vue *statique*), tandis que la transduction apporte un point de vue fonctionnel (*dynamique*).

Définition C.3
Un automate fini est déterministe s'il possède un seul état initial et si 2 transitions issues d'un même état-départ ont des labels distincts. Ainsi, un mot reconnu est représenté par un unique parcours sur le graphe associé à l'automate.

Définition C.4
Un transducteur fini *peut être décrit comme un automate à un seul état initial, à la nuance près que les transitions sont étiquetées non par des lettres, mais par des couples (mot,mot). Un transducteur définit une relation entre les mots "d'entrée" et les mots "de sortie" (valeurs produites), la transduction.*
La transduction réalisée *par un transducteur fini est* rationnelle.

Comme pour la *transduction*, le *transducteur* offre un point de vue *statique* ou *dynamique*.
Dans le premier cas, c'est une machine qui reconnaît des paires de mots données en entrée.
Dans le second cas, la machine lit un mot en entrée et imprime en sortie les mots-images.

Théorème C.1
Dans un monoïde libre, *il y a identité entre l'ensemble des* parties reconnaissables par un automate fini *et l'ensemble des* parties rationnelles.
Soit X et Y deux alphabets; de X^* sur Y^*, l'ensemble des transductions reconnaissables *est strictement inclus dans l'ensemble des* transductions rationnelles.

Nous ne nous intéresserons qu'à une certaine classe de transducteurs, ceux que nous qualifions d'*alphabétiques*, dont les étiquettes sont des couples (lettre,mot).

Définition C.5
L'automate obtenu en "oubliant" les mots de sortie d'un transducteur alphabétique *est appelé automate* sous-jacent *au transducteur.*

Définition C.6
Un transducteur alphabétique *dont l'automate* sous-jacent *est* déterministe *est dit* séquentiel. *Il définit une fonction dite* transduction séquentielle *qui, à une suite d'instructions donnée, associe une suite unique de valeurs produites.*

Définition C.7

Soit AT un automate ou un transducteur. Appelons trajet *un chemin allant d'un état initial à un état* d'acceptation *(état final).*

AT est émondé *si pour chacun de ses états, il existe un* trajet *qui le traverse.*

Définition C.8

Soit AT un automate ou un transducteur.

*Un état-*poubelle *peut être atteint depuis l'état initial, n'est pas* d'acceptation*, et aucune transition ne permet de le quitter. C'est un état* cul-de-sac.

Définition C.9

Une grammaire algébrique est dite pointée *si elle est munie d'un axiome et d'un seul.*

Une grammaire algébrique pointée *est dite* réduite *si et seulement si les 2 conditions suivantes sont satisfaites:*

- *aucun non-terminal n'engendre un langage vide*
- *tous les non-terminaux peuvent être dérivés de son axiome.*

Bibliographie

[AB88] Jean-Michel Autebert and Luc Boasson. *Transductions rationnelles.* Masson, 1988.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principes, Techniques, and Tools.* Addison-Wesley, 1986.

[Aut94] Jean-Michel Autebert. *Théorie des langages et des automates.* Masson, 1994.

[Ber79] Jean Berstel. *Transductions and Context-Free Languages.* Teubner, Stuttgart, Germany, 1979.

[Coh99] Albert Cohen. *Program Analysis and Transformation: from the Polytope Model to Formal Languages / Analyse et transformation de programmes : du modèle polyédrique aux langages formels.* PhD thesis, uvsq, December 1999.

[Gup98] R. Gupta. A code motion framework for global instruction scheduling. In *International Conference on Compiler Construction (CC'98)*, pages 219–233, 1998.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[KRS94] J. Knoop, O. Rüthing, and B. Steffen. Optimal Code Motion: Theory and Practice. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 16 n^o 4, pages 1117–1155. ACM Press, 1994.

[Lan91] Bernard Lang. *Current Issues in Parsing Technology*, chapter Towards a Uniform Formal Framework for Parsing, pages 153–171. Kluwer Academic Publishers, 1991.

[Muc97] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, 1997.

[Niv68] Maurice Nivat. *Transductions des langages de Chomsky*, volume 18. Annales de l'Institut Fourier, 1968.

[RS97] G. Rozenberg and A. Salomaa. *Handbook of formal languages*, volume 1: Word Language Grammar. Springer-Verlag, 1997.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399